A Topological Sorting Algorithm for Large Graphs

DEEPAK AJWANI, University College Cork
ADAN COSGAYA-LOZANO and NORBERT ZEH, Dalhousie University

We present an I/O-efficient algorithm for topologically sorting directed acyclic graphs, called IterTS. In the worst case, our algorithm is extremely inefficient and performs $O(n \cdot \text{sort}(m))$ I/Os. However, our experiments show that IterTS achieves good performance in practice. 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, a standard DFS-based topological sorting built on top of a semiexternal 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.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: External-memory algorithms, graph algorithms

ACM Reference Format:
Ajwani, D., Cosgaya-Lozano, A., and Zeh, N. 2012. A topological sorting algorithm for large graphs. ACM J. Exp. Algor. 17, 3, Article 3.2 (August 2012), 21 pages.
DOI = 10.1145/2133803.2330083 http://doi.acm.org/10.1145/2133803.2330083

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. This problem arises in a number of applications where the goal is to find a linear ordering of items or activities (represented by the vertices of G) consistent with a set of pairwise ordering constraints (represented by the edges of G). The problem of topologically sorting large

Part of this work was carried out while D. Ajwani was a postdoctoral fellow at the MADALGO Center for Massive Data Algorithmics, Aarhus University, Denmark.
D. Ajwani's work was supported by the Danish National Research Foundation and an Enterprise Partnership Scheme grant from the Irish Research Council for Science, Engineering and Technology (IRCSET) and IBM. A. Cosgaya's work was supported by the National Council of Science and Technology of Mexico (CONACyT) and by the Natural Sciences and Engineering Research Council of Canada (NSERC). N. Zeh's work was supported by NSERC and the Canada Research Chairs programme.
Authors' addresses: D. Ajwani, Centre for Unified Computing, Western Gateway Building, University College Cork, Ireland; email: d.ajwani@cs.ucc.ie; A. Cosgaya-Lozano and N. Zeh, Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, NS B3H 1W5, Canada; email: {acosgaya, nzeh}@cs.dal.ca.

DOI 10.1145/2133803.2330083 http://doi.acm.org/10.1145/2133803.2330083

DAGs arises, for example, in the application of recent multiple sequence alignment algorithms [Zhang and Waterman 2003, 2005] to large collections of DNA sequences and as a building block for other algorithms for large datasets.

The efficient processing of large datasets beyond the size of the computer's main memory is feasible only using *I/O-efficient algorithms*. These algorithms aim to minimize the number of disk accesses in computations on large datasets. They are designed and analyzed in the *I/O model* [Aggarwal and Vitter 1988], 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 such operation transfers a block of B consecutive data items to or from disks. The cost of an algorithm in the I/O model is the number of I/Os it performs. One of the earliest results in the I/O model shows that sorting N elements takes $\mathrm{sort}(N) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os [Aggarwal and Vitter 1988]. Note that, for reasonable values of B, M, and N, this is significantly less than one I/O per element.

The development of I/O-efficient graph algorithms even for the most elementary problems, such as topological sorting, poses serious challenges. This is mainly because most traditional graph algorithms are based on graph exploration techniques, such as depth-first search (DFS) and breadth-first search (BFS). These algorithms access the vertices of the graph in an unpredictable fashion and usually perform one I/O per vertex when processing inputs beyond the size of main memory. This is true also for standard topological sorting algorithms covered in undergraduate texts (e.g., Cormen et al. [2001]): One such algorithm repeatedly numbers and removes sources (in-degree-0 vertices). Another algorithm performs a DFS traversal of the graph and numbers the vertices in reverse postorder.

For general directed graphs, no techniques are known for designing algorithms with better I/O complexity than algorithms based on graph exploration. The currently best general directed DFS and BFS algorithms perform $O((n+m/B)\log n)$ I/Os [Buchsbaum et al. 2000], which is efficient for dense graphs but worse than standard internal-memory DFS and BFS for sparse graphs. For special classes of directed graphs, such as planar graphs, a wide range of problems, including topological sorting [Arge and Toma 2004; Arge et al. 2003; Maheshwari and Zeh 2009, 2004], can be solved I/O-efficiently using techniques that exploit the structure of these graphs. Similarly, a number of I/O-efficient algorithms for undirected graphs have been obtained using alternate techniques that do not rely on graph exploration.

One technique that has proven useful for designing I/O-efficient graph algorithms is *time-forward processing* [Chiang et al. 1995]. This technique can be used to solve 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 $\psi(\cdot)$-labels of x's in-neighbors. 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-neighbors. In order to be implemented I/O-efficiently, time-forward processing requires the vertices of the DAG to be given in a topologically sorted order. Since 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. For examples, see Arge et al. [2003], Maheshwari et al. [2008], Arge and Revsbæk [2009], and Maheshwari and Zeh [2008]. A general I/O-efficient topological sorting algorithm would greatly increase the applicability of this technique.

The lack of provably I/O-efficient algorithms for sparse directed graphs has led to the development of a number of heuristic approaches for solving problems on directed graphs I/O-efficiently. Most notably, Sibeyn et al. [2002] proposed a DFS heuristic that performs extremely well if the vertex set of the graph fits in memory but breaks down on larger graphs. Cosgaya-Lozano and Zeh [2009] proposed a contraction-based heuristic for computing the strongly connected components of a directed graph. A different implementation of this general approach to computing strongly connected components was also employed by Laura and Santaroni [2011]. In this article, we propose an algorithm for topologically sorting directed acyclic graphs that falls into this category of efficient heuristics. In the worst case, it performs $O(n \cdot \text{sort}(m))$ I/Os, but our experiments show that it performs well in practice and can efficiently process graphs beyond the reach of existing algorithms, including an algorithm based on the DFS heuristic by Sibeyn et al. [2002].

The rest of this article 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 set-up and results. In Section 5, we give concluding remarks.

2. TOPOLOGICAL SORTING BY ITERATIVE IMPROVEMENT (ITERTS)

Our new topological sorting algorithm, called IterTS throughout this article, is based on the following strategy (see Algorithm 1). 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 $\nu(\cdot)$ and the number m_s of edges in G_ν, we proceed in iterations, each of which updates the numbering $\nu(\cdot)$ with the goal of increasing the number of satisfied edges. The computation of the updated numbering ensures that this numbering satisfies strictly more edges than the previous numbering. Thus, the algorithm is guaranteed to terminate, slowly in the worst case, quickly in practice. The numbering $\nu(\cdot)$ is represented by storing the number $\nu(x)$ with every vertex $x \in V$ and by storing the numbers $\nu(x)$ and $\nu(y)$ with every edge $xy \in E$. Note that this in particular allows us to decide, for every edge $xy \in E$, whether it belongs to G_ν, simply by comparing $\nu(x)$ and $\nu(y)$.

Our description of the algorithm consists of four parts. In Section 2.1, we provide the details of the procedure InitialNumbering we use to compute the initial numbering $\nu(\cdot)$. In Section 2.2, we discuss the procedure ImproveNumbering we use to update the numbering $\nu(\cdot)$ in each iteration. In Section 2.3, we analyze the I/O complexity of the algorithm. In Section 2.4, we discuss a heuristic whose addition to procedure ImproveNumbering led to a tremendous performance improvement by reducing the number of iterations the algorithm needs to find a topological ordering.

ALGORITHM 1: IterTS

Input: A DAG $G = (V, E)$;
Output: A numbering $\nu(\cdot)$ of the vertices in V such that $\nu(x) < \nu(y)$, for every edge $xy \in E$;

```
1 m := |E|;
2 (ν(·), m_s) := InitialNumbering(G);
3 while m_s < m do
4     (ν(·), m_s) := ImproveNumbering(G, ν(·));
```

ALGORITHM 2: InitialNumbering

Input: A DAG $G = (V, E)$;
Output: A numbering $\nu(\cdot)$ of the vertices in V and the number m_s of edges in G satisfied by $\nu(\cdot)$; every vertex $x \in V$ stores its number $\nu(x)$; every edge $xy \in E$ stores the numbers $\nu(x)$ and $\nu(y)$; $\nu(\cdot)$ satisfies at least half of the edges in G;

```
1  T := OutTree(G);
2  L := EulerTour(T);
3  (ν_l(·), ν_r(·)) := PreorderNumberings(L);
4  m_l := number of edges in G satisfied by ν_l(·);
5  m_r := number of edges in G satisfied by ν_r(·);
6  if m_l > m_r then
7  |  return (ν_l(·), m_l);
8  else
9  |  return (ν_r(·), m_r);
```

2.1. Computing the Initial Numbering

We use Algorithm 2 to compute the initial numbering of the vertices of G. First, we compute an out-tree of the source s of G, that is, a spanning tree T of G whose root is s and all of whose edges are directed away from s (Figure 1(a)). This requires G to have only one source s. If this is not the case, procedure OutTree modifies G by adding an edge from the first source s it finds to every source it finds after that; the resulting graph is clearly acyclic, and a topological ordering of this DAG is also a topological ordering of G. Given the tree T, we compute an Euler tour L of T and two preorder numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$, numbering the vertices of G in the order they are first visited by L and its reversal, respectively (see Figures 1(b) and 1(c)). These two numberings satisfy the edges of T because they are preorder numberings of T, and one of them satisfies at least half of the edges of G not in T. We count the number of edges of G satisfied by each numbering and return the numbering that satisfies more edges. To count the number of satisfied edges, we label every edge with the numbers assigned to its endpoints by $\nu_l(\cdot)$ and $\nu_r(\cdot)$. Next, we discuss the individual steps in more detail and argue that each takes O(sort(m)) I/Os. This shows that the initial numbering $\nu(\cdot)$ can be computed using O(sort(m)) I/Os.

2.1.1. Computing the Out-Tree. The procedure OutTree, which augments G so that it has only a single source s (if necessary) and computes an out-tree T of G with root s is shown in Algorithm 3. Assuming G has only one source s to begin with, an out-tree with root s can be obtained by choosing, for each vertex $x \neq s$, an arbitrary in-edge of x to be included in T. This works because G is acyclic. So the algorithm sorts the edges of G by their heads and scans this sorted edge list to add the first in-edge of every vertex in the list to T. In the process, it can identify the vertices that have no in-edges in the list—the sources of G—and add edges from the first source to every subsequent source if necessary to ensure that G has only one source s. Since this procedure sorts the edge set of G once and then scans this sorted edge list once, the cost of this step is O(sort(m)) I/Os.

2.1.2. Computing an Euler Tour. An Euler tour L of T can be computed using O(sort(n)) I/Os using the standard approach of creating two directed copies of each edge of T, sorting these edges by their heads, and scanning this sorted list to identify the successor of each edge in the Euler tour (see Algorithm 4).[1] Once we have created this list L, we apply an I/O-efficient list ranking algorithm [Chiang et al. 1995; Sibeyn 1997] to L and

[1]We chose to describe the search for an in-edge of s in Line 10 as a separate scan of the edge list, in order not to obfuscate the pseudocode. Our actual implementation identifies this edge as part of the preceding loop.

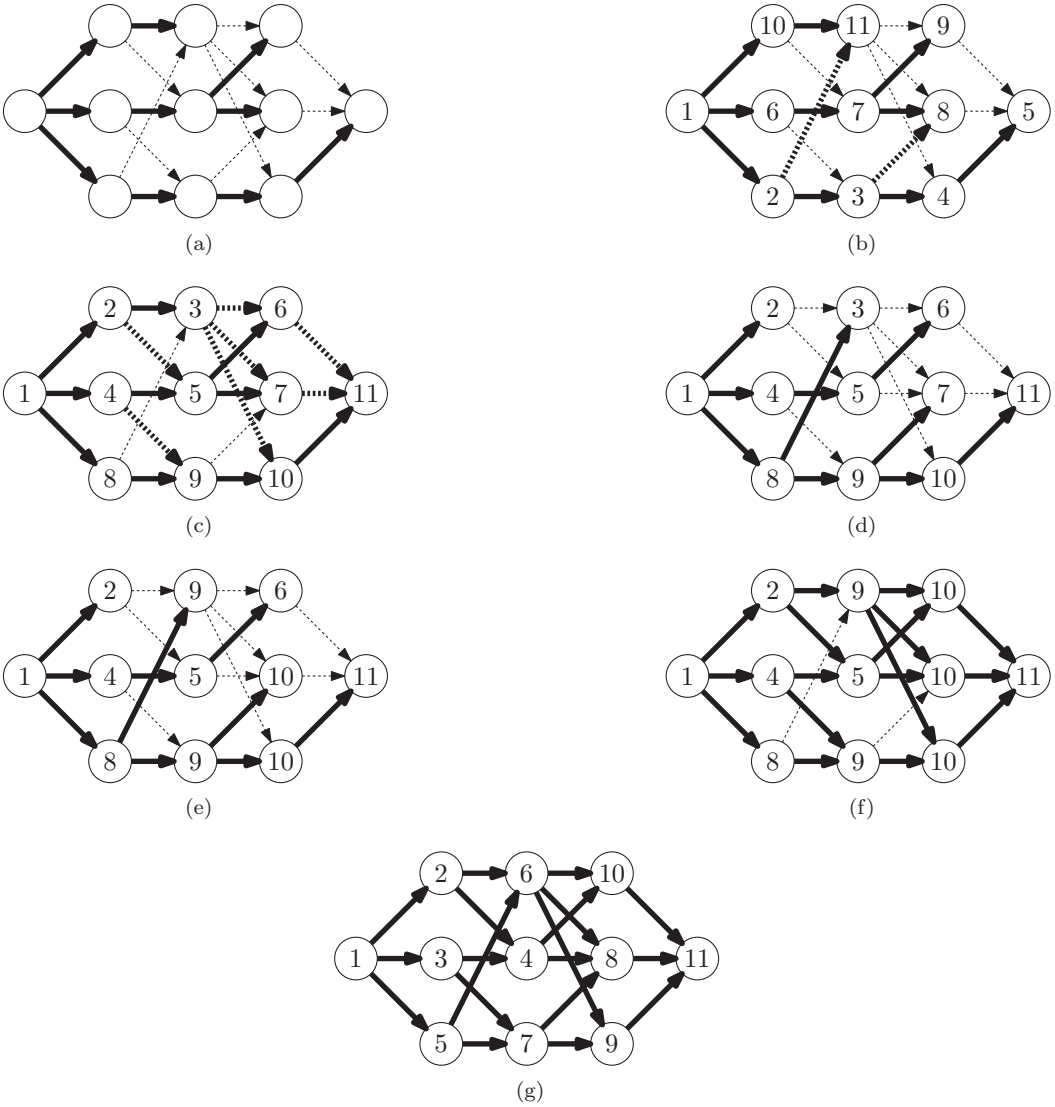


Fig. 1. (a) A DAG with an out-tree T_0 shown using bold edges. (b/c) The two preorder numberings of T_0. The one in (c) satisfies more nontree edges (bold dotted) and is the one we choose as the initial numbering $\nu(\cdot)$. (d) The out-tree T_1 chosen in the first iteration of IterTS. (e) The numbering $\nu'(\cdot)$ computed to satisfy all edges in T_1. (f) The numbering $\nu''(\cdot)$ computed to satisfy all edges in G_ν (bold). (g) The updated numbering obtained by sorting the vertices according to $\nu''(\cdot)$ and then numbering them in order. The subgraph satisfied by this numbering contains all edges of G. Thus, we have a topological ordering of G, and the algorithm terminates.

sort the edges in L by their ranks, that is, in the order they are visited in the Euler tour. This also takes O(sort(n)) I/Os.

2.1.3. Computing Preorder Numberings. Given the edges in the Euler tour L in sorted order, the preorder numbering $\nu_l(\cdot)$ defined by L can be computed using a single scan of L, as shown in Algorithm 5. Observing that the preorder numbering $\nu_r(\cdot)$ of the reverse of L is nothing but the postorder numbering defined by L with vertices numbered

ALGORITHM 3: OUTTREE

Input: A DAG $G = (V, E)$; all vertices in V have positive integer IDs;
Output: G augmented if necessary to ensure it has only one source s; an out-tree T of G with root s;

```
1  sort the vertices in V by their IDs;
2  sort the edges in E by their heads;
3  s := 0;                                          // the first source of G we see, none yet
4  v := 0;                                          // the last vertex whose first in-edge has been added to T
5  for every edge xy ∈ E, in sorted order do
6      if y > v then                                // we haven't seen any in-edge of y yet
7          add edge xy to T;
8          while v < y do                           // vertices between v and y have no in-edges
9              v := next vertex in V;
10             if v < y then
11                 if s = 0 then                    // v is the first source we see ...
12                     s := v;                      // ... remember it
13                 else                             // there already is a source
14                     add the edge sv to T and G;
15 return T;
```

ALGORITHM 4: EULERTOUR

Input: A tree T with root s;
Output: An Euler tour L of T that starts at s;

```
1  L := {xy, yx | xy ∈ T}; mark the edge xy as a forward edge and the edge yx as a
   backward edge, for each edge xy ∈ T;
2  sort the edges in L by their heads;
3  for every edge xy ∈ L, in sorted order do
4      if xy is the first edge with head y in L then
5          e := yx;
6      if xy is the last edge with head y in L then
7          succ(xy) := e;
8      else
9          succ(xy) := yz, where zy is the next edge in L;
10 scan L to find an in-edge xs of s, and set succ(xs) := nil to break the cycle;
11 apply list ranking to the list defined by the succ(·) pointers;
12 sort the elements of L by their ranks;
13 return L;
```

backwards, we can compute both numberings in the same scan, as shown in Algorithm 5. Thus, both numberings can be computed using $O(n/B)$ I/Os.

2.1.4. Counting Satisfied Edges. To count the number of edges satisfied by $\nu_l(\cdot)$ and $\nu_r(\cdot)$, we label the edges of G with the numbers assigned to their endpoints. To this end, we sort the vertices in G by their IDs. Then, we sort the edges by their heads and scan the sorted vertex and edge lists to copy the number $\nu_l(x)$ of each vertex x to every edge that has x as a head. To copy $\nu_l(x)$ to every edge that has x as a tail, for each vertex x, we sort the edges by their tails and scan the sorted vertex and edge lists again. This takes $O(\text{sort}(m))$ I/Os. Now a single scan of the edge list of G suffices to test, for every edge xy, whether $\nu_l(x) < \nu_l(y)$ and increase m_l accordingly. To compute m_r, we create a copy

ALGORITHM 5: PreorderNumberings

Input: An Euler tour L of T;
Output: The two preorder numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$ defined by L and Reverse(L) represented as two lists N_l and N_r of pairs $(x, \nu_l(x))$ and $(x, \nu_r(x))$, respectively;

```
1  N_l := {(s, 1)};
2  N_r := {(s, 1)};
3  l := 2;
4  r := n;
5  for every edge xy ∈ L, in order do
6      if xy is a forward edge then
7          add the pair (y, l) to N_l; l := l + 1;
8      else
9          add the pair (x, r) to N_r; r := r − 1;
10 return (N_l, N_r);
```

ALGORITHM 6: ImproveNumbering

Input: A DAG G and a numbering $\nu(\cdot)$ of its vertices; every vertex x in G stores its number $\nu(x)$; every edge xy stores the numbers of its endpoints;
Output: An updated numbering $\nu(\cdot)$ of the vertices of G, represented in the same way as the input numbering; the number m_s of edges satisfied by the new numbering $\nu(\cdot)$;

```
1  T := MaximallyViolatedOutTree(G, ν(·));
2  ν'(·) := SatisfyTreeEdges(T, ν(·));
3  ν''(·) := SatisfyPreviouslySatisfiedEdges(G, ν'(·));
4  sort the vertices of G by ν''(·);
5  ν(·) := number the vertices of G in order;
6  m_s := the number of edges satisfied by ν(·);
7  return (ν(·), m_s);
```

of the edge list of G and repeat the above procedure with $\nu_r(\cdot)$ instead of $\nu_l(\cdot)$. By doing this, we produce two copies of the edge list of G, one where the edges are labeled with the $\nu_l(\cdot)$ labels of their endpoints and one where they are labeled with the $\nu_r(\cdot)$ labels of their endpoints. After comparing which numbering satisfies more edges, we simply choose the appropriate edge list as part of the input to the first invocation of procedure ImproveNumbering.

2.2. Growing the Satisfied Subgraph

Given the initial numbering $\nu(\cdot)$ computed in Line 2 of procedure IterTS, Lines 3–4 repeatedly invoke procedure ImproveNumbering (Algorithm 6) to update $\nu(\cdot)$ until it satisfies all edges of G, that is, until it is a topological ordering of G. Procedure ImproveNumbering updates $\nu(\cdot)$ in four phases. First, we compute a "maximally violated" out-tree T of G and compute a numbering $\nu'(\cdot)$ such that $\nu'(x) \geq \nu(x)$, for all x, and $\nu'(\cdot)$ satisfies all edges of T. Then, we compute a numbering $\nu''(\cdot)$ such that $\nu''(x) \geq \nu'(x)$, for all x, and $\nu''(\cdot)$ satisfies all edges of the graph G_ν satisfied by $\nu(\cdot)$. Then, we compute the updated numbering $\nu(\cdot)$ by sorting the vertices of G by $\nu''(\cdot)$ and numbering them in order. Finally, we count the number of edges satisfied by this updated numbering $\nu(\cdot)$.

2.2.1. Computing a Maximally Violated Out-Tree. The "maximally violated" out-tree T of G computed in Line 1 of procedure ImproveNumbering contains, for every vertex $x \neq s$, the in-edge yx with maximum tail label $\nu(y)$ (Figure 1(d)). If there is an in-edge of x violated by $\nu(\cdot)$, this edge yx is also violated and needs the largest adjustment of $\nu(x)$ to

ALGORITHM 7: SatisfyGraphEdges

Input: A topologically sorted DAG G and a numbering $\nu(\cdot)$ of its vertices; every vertex x in G stores its number $\nu(x)$ and its position in the topological ordering; the algorithm refers to vertices by their positions in the topological ordering;
Output: A numbering $\nu'(\cdot)$ of the vertices of G, represented in the same way as the input numbering $\nu(\cdot)$; $\nu'(\cdot)$ satisfies all edges of G and satisfies $\nu'(x) \geq \nu(x)$, for all x;

```
1 Q := ∅;                                   // a priority queue to send labels along the edges of G
2 for every vertex x do
3     ν'(x) := ν(x);
4     while the minimum-priority entry (ν'(y), p) in Q has priority p = x do
5         (ν'(y), p) := DeleteMin(Q);
6         ν'(x) := max(ν'(x), ν'(y) + 1);
7     for every out-edge xz of x do
8         insert ν'(x) into Q with priority z;
9 return ν'(·);
```

become satisfied. Such a tree T can be computed using O(sort(m)) I/Os using a simple adaptation of procedure OutTree (Algorithm 3). All that is required is to alter Line 2 to sort the edges of G primarily by their heads and secondarily by decreasing numbers of their tails (and the augmentation of G to enforce that there is a unique source is not needed).

2.2.2. Satisfying Tree Edges. To satisfy the tree edges, we first compute a preorder numbering of T using Algorithm 4 and (a simplification of) Algorithm 5. Then, we sort the vertices of T by their preorder numbers and the edges of T by the preorder numbers of their tails. The result is a topological ordering of T. Thus, we can apply time-forward processing to compute $\nu'(\cdot)$ using O(sort(n)) I/Os. More precisely, we apply the procedure SatisfyGraphEdges, shown as Algorithm 7, to T and the numbering $\nu(\cdot)$. This procedure takes as input a topologically sorted DAG whose vertices are numbered using an input numbering $\nu(\cdot)$. It computes a new numbering $\nu'(\cdot)$ such that, for every vertex x, $\nu'(x)$ is as small as possible while ensuring that $\nu'(x) \geq \nu(x)$, for all x, and $\nu'(\cdot)$ satisfies all edges of the DAG (see Figure 1(e)).

2.2.3. Satisfying the Edges in G_ν. To ensure the edges that were satisfied by $\nu(\cdot)$ before the current iteration remain satisfied, we process G_ν and compute a numbering $\nu''(\cdot)$ from $\nu'(\cdot)$ that satisfies all edges in G_ν (see Figure 1(f)). To compute this numbering, we extract G_ν from G, which takes a single scan because every edge of G stores the numbers assigned to its endpoints by $\nu(\cdot)$. Then, we sort the vertices of G_ν by the numbers assigned to them by $\nu(\cdot)$, and the edges of G_ν by the numbers assigned to their tails. By definition, this is a topological ordering of G_ν. Hence, we can apply procedure SatisfyGraphEdges to G_ν and the numbering $\nu'(\cdot)$ to compute $\nu''(\cdot)$ using O(sort(m)) I/Os.

2.2.4. Numbering Vertices Consecutively and Counting Satisfied Edges. The final step in the computation of the updated numbering $\nu(\cdot)$ is to sort the vertices by $\nu''(\cdot)$ and number them in order, and to count the number of edges $\nu(\cdot)$ now satisfies (see Figure 1(g)). By numbering the vertices in order at the end of each iteration, we ensure that the numbers assigned to the vertices of G do not grow excessively large. We count the number of edges satisfied by the updated numbering $\nu(\cdot)$ in the same way as discussed in Section 2.1.4. Thus, this final step of procedure ImproveNumbering also takes O(sort(m)) I/Os.

2.3. Analysis

As part of the discussion of IterTS, we argued that the initialization and each iteration of the algorithm take O(sort(m)) I/Os. Thus, the I/O complexity of IterTS depends on the number of iterations it 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. In this proof, we refer to the vertex numbering produced by procedure InitialNumbering as $\nu_0(\cdot)$. The numberings $\nu'(\cdot)$, $\nu''(\cdot)$, and $\nu(\cdot)$ computed in the ith invocation of procedure ImproveNumbering are referred to as $\nu'_i(\cdot)$, $\nu''_i(\cdot)$, and $\nu_i(\cdot)$, respectively. In particular, the input to the ith invocation is the numbering $\nu_{i-1}(\cdot)$.

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 $\nu_i(\cdot)$ 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, as claimed by the lemma.

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) ≤ 1.

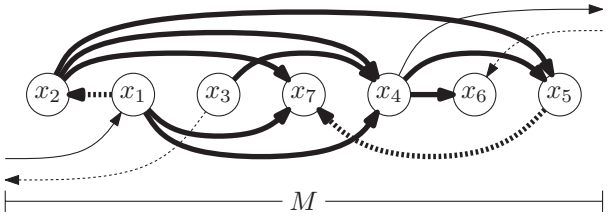
For $i=k>0$, we can inductively assume the claim holds for all $i<k$. To prove the claim for $i=k$, it suffices to prove that $\nu''_k(\cdot)$ satisfies all in-edges of vertices x with l-dist(x) $\leq k+1$ because $\nu_k(\cdot)$ is obtained by ordering the vertices by $\nu''_k(\cdot)$ and then numbering 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-neighbor 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).

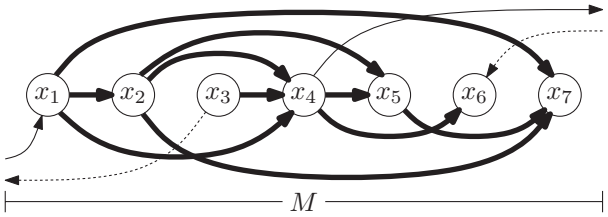
For l-dist(x) $=0$, we have $x=s$ and $\nu''_k(s) = \nu'_k(s) = \nu_{k-1}(s) = 1$ because s is the source of $G_{\nu_{k-1}}$ and the root of the out-tree T_k computed in the kth iteration. For $0 < l$-dist(x) $\leq k$, we have $\nu'_k(x) = \max(\nu_{k-1}(x), \nu'_k(p_k(x))+1)$, where $p_k(x)$ denotes x's parent in T_k. However, we have l-dist($p_k(x)$) $< l$-dist(x) and, hence, by the induction hypothesis, $\nu'_k(p_k(x)) = \nu_{k-1}(p_k(x))$. Furthermore, $\nu_{k-1}(p_k(x)) < \nu_{k-1}(x)$ because $p_k(x)$ is an in-neighbor 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-neighbor 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-neighbor 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-neighbor 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-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 and because the longest path in G has at most $n-1$ edges, IterTS is guaranteed to terminate after at most $n-2$ iterations, that is, it performs O($n \cdot$ sort(m)) I/Os in the worst case. For many graphs, however, 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 practice, IterTS terminates much faster than predicted by Lemma 2.1.



(a) Subgraph before processing.



(b) Subgraph after processing.

Fig. 2. Local topological ordering of a memory-sized subgraph. In (a), vertices are arranged by $\nu''(\cdot)$. Bold edges are in the subgraph, thin edges are not. Solid edges are satisfied, dotted ones are not. In (b), the vertices are rearranged to ensure that all edges local to the subgraph are satisfied.

2.4. Satisfying Local Edges

By our analysis in Section 2.3, 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 "locally" violated edges whose endpoints are "not too far apart" in the current ordering. To this end, we add a step between Lines 4 and 5 of procedure ImproveNumbering. This step starts with the list of vertices sorted according to $\nu''(\cdot)$ and locally rearranges vertices so that all locally violated edges become satisfied. Line 5 then numbers the vertices of G according to this updated ordering.

The details of this local rearrangement step are shown in Algorithm 8. The algorithm greedily breaks the list V of vertices sorted by $\nu''(\cdot)$ into maximal contiguous sublists $V_1, V_2, \ldots, V_q$ such that the vertices in each sublist V_i induce a subgraph $G[V_i]$ that fits in memory. We load each such subgraph $G[V_i]$ into memory and topologically sort it, producing a list V_i' that stores the vertices of V_i in topologically sorted order. This is illustrated in Figure 2. The output of this step is a new vertex list V' obtained by replacing each list V_i in V with V_i'. By definition, the ordering of the vertices in V' satisfies all edges that have both their endpoints in the same sublist V_i'. An edge whose endpoints belong to different sublists V_i' and V_j' is satisfied if and only if it was satisfied already by $\nu''(\cdot)$ because the sublists V_i' and V_j' appear in the same order in V' as the sublists V_i and V_j in V. Thus, reordering the vertices of G in this fashion does not decrease the number of satisfied edges.

The labeling of the edges of G with the $\nu''(\cdot)$-labels of their endpoints in Line 1 of procedure SatisfyLocalEdges can be done using O(sort(m)) I/Os, as discussed in Section 2.1.4. After sorting the vertices and edges using O(sort(m)) I/Os, the remainder of procedure SatisfyLocalEdges takes O(m/B) I/Os. Indeed, the main loop (Lines 7–26) scans the vertex list of G, and the edges of G are sorted by their higher endpoints so that all iterations of the inner loops (Lines 9–11 and 14–16) reduce to two scans of the sorted edge list of G. All other computation happens in memory. In summary, this additional local reordering of the edges increases the cost of procedure ImproveNumbering, that

ALGORITHM 8: SatisfyLocalEdges

Input: A DAG G and a numbering $\nu''(\cdot)$ of its vertices;
Output: A vertex list V' representing an ordering of the vertices of G that satisfies all "local" edges of G and does not violate any edge satisfied by $\nu''(\cdot)$;

```
1  label every edge xy with the numbers ν''(x) and ν''(y) of its endpoints; let
   ν''_l(xy) := min(ν''(x), ν''(y)) and ν''_h(xy) := max(ν''(x), ν''(y));
2  sort the vertices of G by their ν''(·) labels;
3  sort the edges of G by their ν''_h(·) labels;
4  n_f := 0;                          // the ν''(·) label of the first vertex in the current subgraph G[V_i]
5  s := 0;                                                  // the size of the current subgraph G[V_i]
6  V' := ∅;
7  for every vertex x, in sorted order do
       // Determine the size increase of G[V_i] resulting from adding x to V_i
8      s := s + 1;                                                                  // this counts x
9      for every edge e with ν''_h(e) = ν''(x) do
10         if ν''_l(e) ≥ n_f then          // e has its other endpoint in G[V_i] iff ν''_l(e) ≥ n_f
11             s := s + 1;
       // If the result fits in memory, add x and its incident edges to G[V_i]
12     if s ≤ M then
13         load x into memory;
14         for every edge e with ν''_h(e) = ν''(x) do
15             if ν''_l(e) ≥ n_f then
16                 load e into memory;
       // Cannot add another vertex to the current subgraph in memory. Sort it.
17     if s > M or x is the last vertex in G then
18         topologically sort the graph G[V_i] in memory;
19         append the resulting sorted vertex list V'_i to V';
20         if s > M then
21             if x is the last vertex in G then
22                 append x to V';
23             else                                  // x is the first vertex of the next subgraph
24                 clear the memory and load x into memory;
25                 s := 1;
26                 n_f := ν''(x);
27 return V';
```

is, the cost of an iteration of procedure IterTS by O(sort(m)) I/Os, but our experiments showed that the number of iterations is decreased substantially using this additional step.

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 [Cormen et al. 2001]. Using this strategy on top of the semi-external DFS heuristic of Sibeyn et al. [2002], 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 naïve implementation of this strategy requires one random access per edge to test, for each neighbor 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 in by PEELTS. This allows us to identify the sources and sinks to remove in each peeling round by scanning this sorted list instead of using random accesses. To arrange the vertices in such an order, we compute an out-tree T of the source using Algorithm 3 and label the vertices of G with their in- and out-degrees in G and with their depths in T. Given T, this information can be computed using the Euler tour technique and list ranking, similar to the computation of preorder numbers using Algorithm 5 [Chiang et al. 1995]. 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 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}^+, respectively. 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

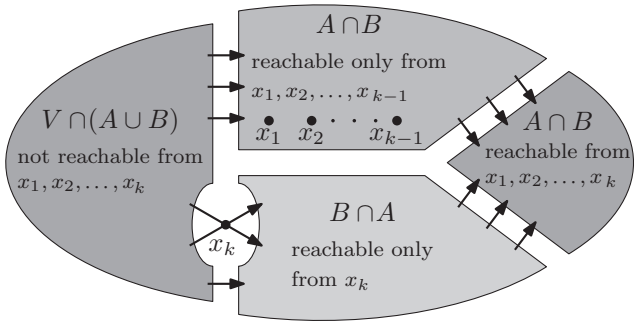


Fig. 3. The five parts of G REACHTS recurses on.

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)

Schudy [2008] proposed a parallel divide-and-conquer algorithm for topological sorting based on reachability queries. 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 (Figure 3). The correctness of this strategy was shown by Schudy [2008]. He also showed that the expected size of each set is $n/2$, making this algorithm terminate after 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 BFS. 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-neighbors 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 details of our implementation of the algorithms discussed in Sections 2 and 3, the environment and datasets 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 [Dementiev et al. 2007], which is an implementation of the C++ STL for external memory computations. For SETS, we used an implementation provided by Andreas Beckmann [2005]. The code for ITERTS, PEELTS, and REACHTS has been submitted along with this article.[2]

Our code is mostly a direct implementation of these algorithms, using STXXL vectors to store the vertex and edge lists of the graph, the STXXL sorting routine to sort the vertices and edges, and the STXXL priority queue to implement the time-forward processing procedure in Algorithm 7. There are a few optimizations we made that are worth discussing here.

4.1.1. External Representation of Vertices and Edges. Throughout the algorithm, we represent every vertex using a 12-byte record of its vertex ID and two integer labels; edges are represented using 16 bytes, an ID and an integer label per endpoint. Specifically, an edge never needs to store more than the IDs of its endpoints and their numbers in the currently relevant vertex numbering. Vertices, on the other hand, need to store two numbers in some of the steps. For example, each vertex x stores both its preorder numbers $\nu_l(x)$ and $\nu_r(x)$ during the computation of the initial vertex numbering $\nu(\cdot)$. Similarly, the computation of $\nu''(\cdot)$ from $\nu'(\cdot)$ in each invocation of procedure IMPROVENUMBERING requires each vertex x to store its current number $\nu(x)$, in order to arrange the vertices of G_v in topologically sorted order, and its number $\nu'(x)$, as a basis for the computation of $\nu''(x)$.

4.1.2. Internal-Memory Topological Sorting and Graph Representation. For topologically sorting the memory-sized subgraphs $G[V_i]$ in the local reordering step of the algorithm discussed in Section 2.4, we used the standard DFS-based topological sorting algorithm [Cormen et al. 2001]. The effectiveness of this local reordering step to speed up the convergence of $\nu(\cdot)$ to a topological ordering depends on the size of the subgraphs $G[V_i]$ sorted in memory in this step. The bigger these graphs are, the faster $\nu(\cdot)$ will converge to a topological ordering. Therefore, we used a fairly compact graph representation to maximize the number of vertices and edges we can fit in memory. This representation consists of a vertex array and an edge array; the edge array stores the concatenation of the adjacency lists of the vertices. Since the DFS requires access only to the out-edges of each vertex, only those edges were stored in the vertex's adjacency list. Moreover, since the tail of every edge in a vertex x's adjacency list is x, there was no need to store the tails of edges, and we represented every edge as a single integer representing its head. By referring to each vertex by its position in the current vertex list of G, there was also no need to store vertex IDs explicitly because the vertices in each graph $G[V_i]$ are consecutive in this list, allowing easy conversion between their IDs and their positions in the in-memory vertex array of $G[V_i]$. This allowed us to represent every vertex using two integers: the index in the edge list of the first edge in its adjacency list, and its number in the topological ordering of $G[V_i]$ we compute. Once all vertices have been numbered by the topological sorting algorithm, we sort them by their numbers in memory and write them back to disk in the resulting order.

Since we store edges in a different order in memory than on disk, it was necessary to sort the edges by their tails before constructing the compact internal-memory representation of $G[V_i]$. During this sorting step, we used a full representation of each edge using both its endpoints, and we dropped the tail of each edge once the edges were

[2]Since the code for SETS was provided to us by Andreas Beckmann and, thus, is not our own work, we did not feel comfortable including it in this submission. Instead, we ask the interested reader to request the code directly from him (beckmann@cs.uni-frankfurt.de).

sorted. Since the purpose of using the compact graph representation was to maximize the size of the subgraphs we can fit in memory, we chose to greedily construct maximal graphs that fit in memory using the compact representation. As a result, the sorting of edges by their tails using the full edge representation involved more data than fits in memory, and this sorting step was accomplished using the STXXL external sorting algorithm. Note, however, that each such sorting step sorts at most $2M$ data and, thus, takes $O(M/B)$ I/Os to complete. In other words, the total cost of these sorting steps is linear.

4.1.3. List Ranking. As part of their implementation of an undirected BFS algorithm, Ajwani et al. [2007] developed an efficient implementation of Sibeyn's [1997] list ranking algorithm based on STXXL. We used this implementation for all the list ranking steps in our algorithm.

4.1.4. (No) Pipelining. STXXL provides a pipelining feature, which allows sequences of scanning steps to be chained together without incurring any I/Os for reading and writing intermediate data streams. To this end, the scanning steps are run simultaneously, and the output records of each scanning step are passed directly to the next step without writing intermediate results to disk. This has the potential to reduce the I/O volume substantially, speeding up the algorithm to an equal degree.

The effectiveness of the pipelining approach depends on the number of scanning steps that can be "collapsed" in this fashion. Scanning steps with sorting steps in between them cannot be pipelined, which is the reason why pipelining is mostly ineffective in our algorithm: The algorithm heavily interleaves sorting and scanning steps and, thus, does not seem amenable to pipelining. Therefore, we mostly did not use pipelining in our algorithm. A notable exception is our implementation of the preorder numbering step (Algorithms 4 and 5), which is heavily pipelined.

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). This limited the input size required to exceed the main memory size and, thus, reduced the time required to run our experiments. All of our timing results refer to wall clock times in hours.

4.3. Datasets

We tested the algorithms on synthetic graphs 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 Graphs. 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 Graphs. 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 Graphs. 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-neighbor in the previous layer and a random out-neighbor in the next layer. Then, we added more random edges between adjacent layers to increase the edge count to m.

Semi-Layered Graphs. 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 Graphs. 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-neighbor and one out-neighbor 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 Graphs. 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.

Web Graphs. 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 priori knowledge of the graph structure. Table I 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: (i) Each algorithm was given at least 10 times as long to process a given input as it took ITERTS to process the same input. If it did not produce a result in the allotted time, we terminated it. This is indicated by dashes in the table, with superscripts indicating the amount of time given to the algorithm. (ii) If ITERTS took more than one day to process an input and was consistently faster than the other algorithms on smaller inputs of the same type, we did not run the other algorithms on this input. This is indicated by stars in the table. (iii) Since SETS is a semi-external algorithm and 2^{26} vertices do not fit in 1GB of memory,[3] we did not run it on larger inputs if it did not finish in the allotted 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 2 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

[3]See Beckmann [2005] for a discussion of the main memory requirements of the SETS implementation.

Table I. Experimental Results

Dashes indicate inputs that could not be processed by the algorithm in the allotted 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.

				IterTS		PeelTS	ReachTS	SeTS
Graph class	n	m	m/n	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}	***
Web	18.5m	298.1m	16.1	3	1.88	—^{1}	7.75	1.30
Web	41.3m	1,150.7m	25.9	4	9.06	—^{4}	—^{4}	3.49
Web	118.1m	1,019.9m	8.6	4	10.13	—^{4}	—^{4}	—^{4}

amount of time, that is, IterTS outperformed them by at least 1 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 four 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 (assuming the vertex IDs are essentially random). 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 2 orders of magnitude faster than IterTS. This concurs with the discussion by Sibeyn et al. [2002], 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.

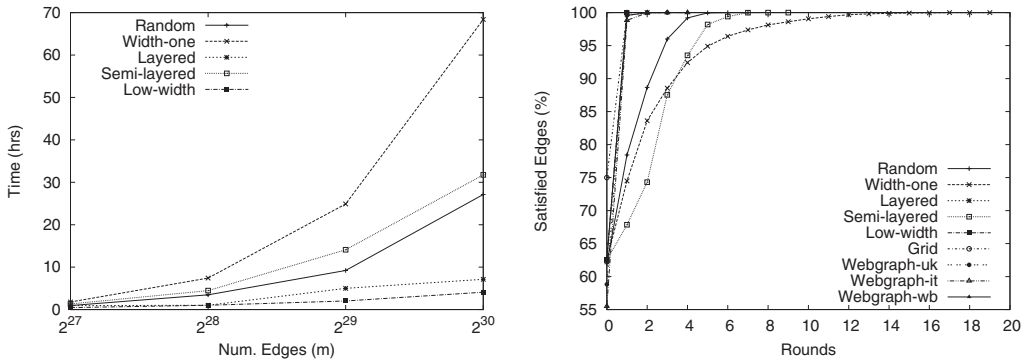
In summary, we conclude that SeTS continues to be the fastest algorithm for semi-external inputs, while IterTS is the clear winner when the vertices no longer fit in memory. PeelTS and ReachTS were not competitive with either SeTS or IterTS.

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 (2,898 to 8,194) 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 to 5s per peeling round.

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



(a) (b)

Fig. 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 $m = 2^{30}$.

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. Therefore, 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 nearly 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 to 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 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 1 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

We developed a new topological sorting algorithm, IterTS, designed to be able to sort DAGs whose vertex sets are beyond the size of main memory and compared it to three reasonable competitors: PeelTS, ReachTS, and SeTS. 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 achieves the goal of efficiently processing graphs whose vertex sets are beyond the main memory size, and it is the first algorithm to do so. On the other hand, SeTS remains the fastest algorithm for topologically sorting inputs whose vertex sets fit in memory.

While IterTS is able to efficiently process larger graphs than any of the competitors we compared it to, its running time on the inputs we tried is still substantial. This should not be surprising, as it repeatedly sorts the input and repeatedly applies the even more costly time-forward processing technique. The key to reducing the cost per iteration therefore is to avoid the use of time-forward processing and to replace it with less costly methods. This is the focus of ongoing efforts on our part. The use of time-forward processing and the internal-memory topological sorting algorithm used in the local reordering step of Section 2.4 are also the only inherently sequential parts of our algorithm. Thus, by eliminating time-forward processing from our algorithm, we will also make progress towards utilizing multiple processor cores to speed up the internal-memory computation of our algorithm. Finally, as already mentioned in Section 4.4.3, the development of a method for topologically sorting DAGs, given a numbering that satisfies almost all its edges, would allow a significant reduction of the number of iterations of the algorithm and would lead to a corresponding improvement of the running time.

REFERENCES

Aggarwal, A. and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Comm. ACM 31,* 9, 1116–1127.

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

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

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

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

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

Beckmann, A. 2005. Parallelizing semi-external depth first search. M.S. thesis, Martin-Luther-Universität, Halle, Germany.

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

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

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. MIT Press.

COSGAYA-LOZANO, A. AND ZEH, N. 2009. A heuristic strong connectivity algorithm for large graphs. In *Proceedings of the 8th International Symposium on Experimental Algorithms*. Lecture Notes in Computer Science, vol. 5526, Springer-Verlag, 113–124.

DEMENTIEV, R., KETTNER, L., AND SANDERS, P. 2007. STXXL: Standard library for XXL data sets. *Software. 38,* 6, 589–637.

LAURA, L. AND SANTARONI, F. 2011. Computing strongly connected components in the streaming model. In *Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems*. Lecture Notes in Computer Science, vol. 6595, Springer-Verlag, 193–205.

MAHESHWARI, A., SMID, M. H. M., AND ZEH, N. 2008. I/O-efficient algorithms for computing planar geometric spanners. *Computat. Geometry 40,* 3, 252–271.

MAHESHWARI, A. AND ZEH, N. 2004. I/O-optimal algorithms for outerplanar graphs. *J. Graph Algor. App. 8,* 1, 47–87.

MAHESHWARI, A. AND ZEH, N. 2008. I/O-efficient planar separators. *SIAM J. Comput. 38,* 3, 767–801.

MAHESHWARI, A. AND ZEH, N. 2009. I/O-efficient algorithms for graphs of bounded treewidth. *Algorithmica 54,* 3, 413–469.

SCHUDY, W. 2008. 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*. 146–151.

SIBEYN, J., ABELLO, J., AND MEYER, U. 2002. Heuristics for semi-external depth-first search on directed graphs. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*. 282–292.

SIBEYN, J. F. 1997. From parallel to external list ranking. Tech. rep. MPI-I-97-1-021, Max Planck Institut für Informatik, Saarbrücken, Germany.

ZHANG, Y. AND WATERMAN, M. S. 2003. An Eulerian path approach to global multiple alignment for DNA sequences. *J. Comput. Bio. 10*, 803–820.

ZHANG, Y. AND WATERMAN, M. S. 2005. An Eulerian path approach to local multiple alignment for DNA sequences. *Proc. Nat. Acad. Sci. 102,* 5, 1285–1290.

Received April 2011; revised November 2011; accepted May 2012