I/O-Efficient Algorithms for Topological Sort and Related Problems

NAIREN CAO, JEREMY T. FINEMAN, KATINA RUSSELL, and EUGENE YANG,
Georgetown University, USA

This article presents I/O-efficient algorithms for topologically sorting a directed acyclic graph and for the more general problem identifying and topologically sorting the strongly connected components of a directed graph $G = (V, E)$. Both algorithms are randomized and have I/O-costs $O(sort(E) \cdot \text{poly}(\log V))$, with high probability, where $sort(E) = O(\frac{E}{B} \log_{M/B}(E/B))$ is the I/O cost of sorting an $|E|$-element array on a machine with size-B blocks and size-M cache/internal memory. These are the first algorithms for these problems that do not incur at least one I/O per vertex, and as such these are the first I/O-efficient algorithms for sparse graphs. By applying the technique of time-forward processing, these algorithms also imply I/O-efficient algorithms for most problems on directed acyclic graphs, such as shortest paths, as well as the single-source reachability problem on arbitrary directed graphs.

CCS Concepts: • **Theory of computation** → **Graph algorithms analysis**; *Sorting and searching;*

Additional Key Words and Phrases: I/O efficient, graph algorithm, topological sort, external memory, strongly connected components

ACM Reference format:
Nairen Cao, Jeremy T. Fineman, Katina Russell, and Eugene Yang. 2022. I/O-Efficient Algorithms for Topological Sort and Related Problems. *ACM Trans. Algorithms* 18, 1, Article 5 (January 2022), 24 pages.
https://doi.org/10.1145/3418356

1 INTRODUCTION

Ullman and Yannakakis [25] and Chiang et al. [10] initiated the study of graph algorithms in the ***I/O model*** [2] over 20 years ago. Despite decades of research and many efficient algorithms for undirected graphs, there are essentially no I/O-efficient algorithms known for even the most basic problems on sparse *directed* graphs. Perhaps the most coveted is an algorithm for topologically sorting a **directed acyclic graph** (**DAG**). A topological sort of a DAG $G = (V, E)$ is an ordering of the vertices such that for every edge $(u, v) \in E$, u precedes v in the ordering.

This article presents the first algorithm for topologically sorting a DAG that is I/O efficient even for sparse graphs. Not only is topologically sorting a fundamental problem on DAGs, but it is also a key subroutine in another general I/O-efficient technique known as time-forward processing [4, 10]. Due to the lack of a good general-purpose algorithm for topological sort, time-forward

This research was supported in part by NSF grants no. CCF-1314633, no. CCF-1617727, and no. CCF-1718700.
Authors' address: N. Cao, J. T. Fineman, K. Russell, and E. Yang, Georgetown University, Department of Computer Science, Saint Mary's Hall, 37th and O Streets, NW, Washington, DC 20057-1232 USA; emails: nc645@georgetown.edu, {jfineman, katina.russell}@cs.georgetown.edu, eugene@ir.cs.georgetown.edu.

1549-6325/2022/01-ART5 $15.00
https://doi.org/10.1145/3418356

processing has only generated provably good results for restricted graph classes such as planar graphs [7, 17, 19].

1.1 The I/O Model and Common Subroutines

The I/O model [2], also called the external-memory model or disk-access-machine model, is a standard theoretical model for understanding the performance of algorithms on large datasets by capturing some notion of locality. The I/O model [2] is a two-level memory hierarchy comprising a size-M ***cache*** (also called internal memory) and an external memory of unbounded size. All data, both in cache and in external memory, is organized in size-B chunks called ***blocks***, so the cache consists of $M/B \geq 1$ blocks. Computation may only occur on data residing in the cache, meaning that data must be transferred from the external memory to cache when needed. These data transfers are performed at the granularity of blocks; each block transfer is called an ***I/O***. The cost of an algorithm in the I/O model, often called the ***I/O cost***, is the number of I/Os performed. Computation itself is free.

The following are common examples of bounds in the I/O model. Iterating over a size-N array in order (assuming $N > M$) has I/O cost $scan(N) = \Theta(N/B)$. Sorting [2] a size-N array has I/O cost $sort(N) = \Theta(\frac{N}{B} \log_{M/B}(N/B))$. A key separation between the RAM model and the I/O model is the difference in cost between models for permuting. In the RAM model, permuting an array is as cheap as scanning. In the I/O model, for most settings of machine parameters permuting is generally as expensive as sorting. Specifically, permuting has I/O cost $\Theta(\min\{N, sort(N)\})$ [2], which for typical values resolves to the sort bound. (The N term corresponds to foregoing an I/O-efficient algorithm entirely—simply run the RAM algorithm and pay an I/O for every operation.) The cost of sorting thus often serves as a lower bound on the I/O cost for problems that can be solved in linear time in the RAM model. Many basic graph problems on sparse graphs (directed or undirected), including topological sort, have $\Omega(sort(V))$ lower bounds in the I/O model [10].

Topological sort. There are two classic linear-time algorithms for topological sort in the RAM model, either repeatedly peeling off the vertices with in-degree 0, or performing a depth-first search and outputting the vertices in reverse order of finish time [14]. The best I/O algorithms known are based on the depth-first search approach, for which there are two algorithms. Neither is efficient for sparse graphs. Chiang et al. [10] provide an algorithm with I/O cost $O(V + sort(E) + \frac{VE}{MB})$, and Buchsbaum et al. [9] give an algorithm with I/O cost $O((V + \frac{E}{B})\log(V/B))$. Both of these bounds include at least a cost of $|V|$, indicating that the algorithm may have to perform a random access or I/O for each vertex. For sparse graphs, notably $|E| = \Theta(V)$, both of these algorithms are *worse* than simply running an ordinary RAM DFS and paying an I/O for every operation.

Time-forward processing. Time-forward processing, originally described by Chiang et al. [10], is a technique that allows for efficient evaluation of circuit-like problems on DAGs. Each vertex (or edge) starts with some value $w(v)$ or $w(u, v)$. The goal is to compute some label $L(v)$ on each vertex, where $L(v)$ depends only on the initial values w and the labels $\{L(u) | (u, v) \in E\}$ on v's immediate predecessors. If the graph is topologically sorted, and certain technical restrictions are met on the function being computed, then the DAG circuit evaluation can be performed in $O(sort(E))$ I/Os by time-forward processing [4, 10]. The first solution [10] has additional restrictions on the relative size of the cache, but Arge's [4] solution removes those restrictions by solving the problem with an I/O-efficient priority queue called a Buffer Tree.

One challenging aspect about graph problems in the I/O model is that vertices cannot generally be processed one by one without sacrificing I/O efficiency. Instead, vertices must be processed (roughly speaking) in parallel by applying various sort and scan steps. Time-forward processing is useful in part because it simulates the effect of processing vertices one by one. Thus, information can propagate arbitrarily far in the graph, provided that the graph is topologically sorted.

1.2 Results

This article gives the following results, all having I/O cost $O(sort(E) \cdot \log^5 V)$, with high probability, on a graph $G = (V, E)$. For conciseness, we assume throughout that $|E| = \Omega(V)$.

— (Sections 3 and 4) A randomized algorithm for topologically sorting a DAG.
— (Section 5) A randomized algorithm for identifying and topologically sorting the **strongly connected components** (**SCCs**). Although this result subsumes topologically sorting a DAG, the algorithm includes additional complications and is thus presented separately.
— Using the topological sort algorithm coupled with time-forward processing [4, 10] yields efficient solutions to other problems on DAGs, such as shortest paths, with the same I/O cost.
— Again applying time-forward processing [4, 10], the SCC algorithm implies a solution to the single-source reachability problem on directed graphs. Specifically, given a directed graph (not necessarily acyclic) and source vertex s, the set of vertices reachable from s can be be identified in $O(sort(E) \cdot \log^5 V)$ I/Os, with high probability.

1.3 Overview of the Approach

The general approach adopted here for a topological sort, loosely based on the IterTS algorithm described by Ajwani et al. [3], is as follows. Initially assign each vertex v a label $L(v)$. Those labels induce an ordering over vertices. (For both our algorithm and IterTS, the labels correspond to a permutation of vertices, but in principle there could be ties.) Adopting the terminology from [3], an edge (u, v) is ***satisfied*** if $L(u) < L(v)$ and ***violated*** otherwise. The goal is to adjust labels over time such that eventually all edges are satisfied.

To understand what makes the problem difficult, consider the following naive realization of the general strategy. Use $L_i(v)$ to denote the label of v in round i. Initially assign all vertices v the label $L_0(v) = 0$. In each round i, update every vertex v's label to $L_i(v) = \max\{L_{i-1}(u) + 1 | (u, v) \in E\}$. (This type of update can be implemented by standard techniques, obtaining the updated label for all vertices via a constant number of sorts.) Although v's label increases to ensure that $L_i(v) > L_{i-1}(u)$, the edge (u, v) only becomes satisfied if $L_i(v) > L_i(u)$; if u's label also increases during this round, then the edge may not be satisfied. In fact, with this algorithm the edge (u, v) would only become satisfied during the round ℓ for ℓ equal to the length of the longest path to u. The end result is an algorithm with $O(V \cdot sort(E))$ worst-case I/O cost. Granted, this realization is particularly naive, but it seems difficult to beat. Indeed, IterTS [3], which applies heuristics to achieve good performance in practice, encounters this bottleneck.

Note that it is trivial to satisfy roughly half the edges immediately by randomly permuting all the vertices and labeling vertices by their rank in the permutation. The challenge is in improving the labeling beyond that point.

1.3.1 *Algorithm Overview.*

An issue with the naive algorithm is that, in some sense, its label updates are too aggressive. Perhaps counter-intuitively, directly ensuring that $L_i(v) > L_{i-1}(u)$ for all edges does not seem to lead to efficient algorithms. Instead, our algorithm temporarily gives up on satisfying certain edges, which makes it easier to satisfy the other edges.

Our algorithm (described more fully in Section 3) performs the following type of recursive partitioning of the graph, somewhat inspired by [12]. Each vertex chooses a random priority value. That priority is propagated some (random) distance forward in the graph. Each vertex adopts the highest priority it has seen, with potentially many vertices adopting the same priority. (This step is performed in a way that ensures that the endpoints of already-satisfied edges remain in priority order.) Next, vertices are collected into groups where all vertices in the group have equal priority. The groups are ordered in increasing order of priority, and finally the algorithm recurses on the vertex-induced subgraphs for each group.

The analysis considers any particular violated edge (u, v). The main claim is that in one full execution of this recursive algorithm, (u, v) has at least a constant probability of becoming satisfied. Repeating the recursive algorithm a logarithmic number of times gives the high-probability result for all edges.

The proof itself is counter-intuitive but also simple in hindsight. Consider a particular violated edge (u, v). Initially, both u and v are in the same recursive subproblem. Ties on priority are good in the sense that they keep u and v in the same recursive subproblem. Eventually, at some recursive step, u and v adopt different priorities and are placed in different recursive subproblems, which fixes the status of (u, v) for the remainder of the execution; the edge becomes satisfied if u's subproblem is ordered before v's, and the edge is said to be ***broken*** if v's subproblem is ordered before u's. The two key components of the analysis are the following: (1) at each level of recursion, the probability that the edge becomes broken is proportional to $1/K$, where K is the number of distances selected from, and (2) after enough levels of recursion, the edge is very likely to cross subproblem boundaries. By selecting distances randomly from a large-enough range, the probability of an edge becoming broken is low enough that the edge is likely to cross subproblem boundaries before it has too many chances of becoming broken. If the edge crosses subproblem boundaries but is not broken, then it must be satisfied.

Extension to strongly connected components. The extended algorithm propagates priorities both backwards and forwards, contracting groups of vertices reached in both directions. The analysis follows a similar framework, but the presence of cycles complicates various aspects of the algorithm and analysis.

Roadmap. The remainder of the article is organized as follows. Section 2 presents some related work on I/O-efficient graph algorithms as well as non-I/O algorithms that use similar techniques. Section 3 presents the algorithm for topological sort, and Section 4 analyzes that algorithm. Section 5 gives the algorithm for strongly connected components and its analysis.

2 RELATED WORK

There is a large body of work, e.g., [1, 5, 6, 9–11, 18, 20–23] on graph algorithms in the I/O model. See [26] or [27] for good surveys on the topic. For undirected graphs, many problems can be solved in $O(sort(E))$ I/Os. (In fact, for dense graphs the logarithmic term in the sort bound can be improved slightly through sparsification [15].) For example, connectivity problems such as connected components, minimum spanning forest, biconnectivity, and so on, can all be solved in $O(sort(E))$ I/Os [10], with high probability. If randomization is not allowed, there are several deterministic algorithms [1, 5, 10, 18, 23], which tend to be at worst logarithmic factors from the sort bound.

The directed analog of the connectivity problems are the reachability problems such as single-source reachability, topological sort, and strongly connected components. The best known bounds for these problems are significantly worse than for their undirected counterparts. Specifically, all of the best existing algorithms [9, 10] have a $|V|$ term in their I/O cost, which is not I/O efficient in general. If the graphs are restricted to be planar graphs, however, many of these problems and more can be solved in $O(sort(E))$ I/Os [5, 8, 19].

Due to the lack of provably efficient algorithms for topological sort, some research has focused on engineering practically efficient algorithms [3, 28]. For example, Ajwani et al. [3] use an iterative approach that follows the same general strategy as our algorithm.

ALGORITHM 1: I/O-Efficient Topological Sort

```
1: function TopologicalSort($G = (V, E)$)
2:     repeat until the vertices $V$ are topologically sorted
3:         RecurTS($G, 1, |V|, 0$)
4: function RecurTS($G, i, j, depth$)                ▷ Reorders the subarray $V[i\,..\,j]$ of vertices
5:     $G' = G[i\,..\,j]$
6:     if $depth \geq \lambda$ or $i = j$ then return
7:     $d_{\max} = (\lambda - depth) \cdot K$
8:     $d_{\min} = d_{\max} - K$
9:     Choose $d$ uniformly at random from $[d_{\min}, d_{\max})$
10:    Choose a uniformly random permutation of priorities $\{\rho(v)\}$
11:    For all $v$, compute $l(v) = \max\{\rho(u) : u \preceq_d v \text{ in } G'\}$
12:    Sort vertices $V[i\,..\,j]$ lexicographically by $\langle l(v), index(v)\rangle$
13:    Partition $V[i\,..\,j]$ into maximal groups $[i_1, j_1], [i_2, j_2], \ldots, [i_t, j_t]$ of a single label ($i_r = j_{r-1} + 1$)
14:    for each $r = 1$ to $t$
15:        RecurTS($G, i_r, j_r, depth + 1$)
```

Related work beyond I/O algorithms. In the RAM model, SCCs can be identified in linear time [14, 24] by performing depth-first search.

Our algorithm shares some similarities with other topological-sort or SCC algorithms that perform recursive decompositions of the graph [12, 13, 16] instead of depth-first search. Coppersmith et al. [13] describe a randomized divide-and-conquer algorithm for computing the strongly connected components that runs in $O(E \log V)$ expected time in the RAM model. Cohen et al. [12] use a labeling scheme, which has a similar recursive structure, to solve an incremental topological sort where edges are added to the graph over time. Fineman's [16] parallel algorithm, which also starts from similar ideas, solves the static reachability problems with $O(E \cdot \text{poly}(\log \text{V}))$ work and $O(V^{2/3} \cdot \text{poly}(\log \text{V}))$ span/depth, with high probability.

The recursive structure of our topological-sort algorithm is most similar to that of Cohen et al. [12] in that the subproblems are defined by performing forward searches from each vertex. Like Fineman's algorithm [16] but unlike the others, our algorithm performs the label propagation/graph search to a bounded distance, but the specific notions of distance are different. Many of the specific details, such as how distances are chosen, also resemble features in Fineman's algorithm [16]. This fact should not be surprising given that there are relationships between parallel algorithms and I/O algorithms (see, e.g., [10] for discussion).

Though there are some similarities in the details between the parallel algorithm [16] and the I/O algorithm presented herein, these similarities are somewhat superficial; the primary challenges in each setting are actually quite different. Notably, our I/O-efficient algorithm leverages time-forward processing, which is not efficient in the parallel model. In contrast, the parallel algorithm strongly exploits random accesses, which are not efficient in the I/O model.

3 TOPOLOGICAL SORT ALGORITHM

This section describes the algorithm for topologically sorting a directed acyclic graph $G = (V, E)$. The algorithm is analyzed in Section 4.

The graph is initially provided with the vertices in arbitrary order. As is typical for I/O algorithms, the graph representation is an array V of vertices and an array E of edges. The records of vertices and edges are as follows. Each vertex is represented by a unique ID, and each edge is represented by the IDs of its endpoints. Because the algorithm will sort the edge array many times, there need not be any assumption on the initial ordering or grouping of edges.

The goal of the algorithm is to gradually reorder vertices such that all edges are eventually satisfied, defined next. For each vertex, $index(v)$ denotes the index of v in the vertex array, i.e., $v = V[i]$ means $index(v) = i$.

Definition 3.1. An edge $(u, v) \in E$ is ***satisfied*** if the $index(u) < index(v)$ in the current vertex ordering. Otherwise, (u, v) is ***violated***.

The algorithm is designed to ensure that once an edge becomes satisfied, it remains satisfied for the rest of the execution.

Algorithm 1 presents a high-level description of the algorithm, ignoring the low-level details necessary to transform the algorithm to an I/O-efficient realization. The main algorithm topologically sorts the graph by performing a sequence of executions of a recursive algorithm, called RecurTS. The goal with each execution of the recursive algorithm is to reorder vertices to satisfy some, ideally a constant fraction, of the violated edges. The main algorithm terminates when all edges have been satisfied, i.e., when the vertices in V are in topological-sort order. Section 3.1 describes RecurTS in more detail, and Section 3.2 briefly describes how to make RecurTS I/O efficient.

3.1 The Recursive Algorithm

At a high level, the recursive algorithm chooses random priorities for each vertex, propagates the priorities some distance in the graph, reorders vertices according to the highest priority observed, and finally recurses on subgraphs induced by vertices of the same priority. Before describing the algorithm in more detail, we first clarify the notion of distance adopted by the algorithm.

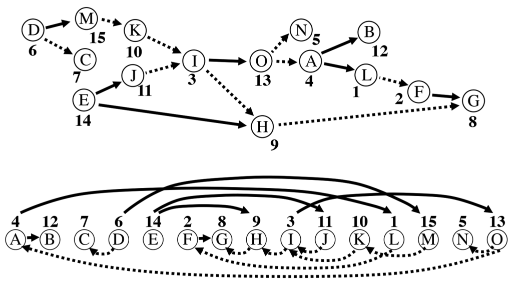
Distances. All distances discussed in this article are with respect to the number of violated edges, i.e., interpreting violated edges as having weight 1 and satisfied edges as having weight 0. If there exists a path from u to v that includes at most d violated edges, then we say u ***can reach*** v ***at distance*** d, denoted $u \preceq_d v$. We also say that u ***is a*** d***-hop predecessor of*** v.

The relation $\preceq_\infty$ is the standard notion of reachability, and when the vertices are in topological-sort order $\preceq_0$ and $\preceq_\infty$ are equivalent. Note that unlike $\preceq_\infty$, when d is a finite fixed distance $\preceq_d$ is not transitive; however, $x \preceq_{d_1} y$ and $y \preceq_{d_2} z$ implies $x \preceq_{d_1+d_2} z$.

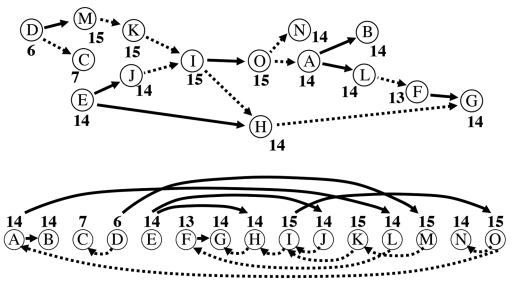
Vertex-induced subgraphs. Each recursive call operates on a contiguous subarray of vertices and the subgraph induced by those vertices. We use $G[i\,..\,j]$ to denote the vertex-induced subgraph of G, induced by vertices in $V[i\,..\,j]$.

Global parameters. The algorithm is parameterized by K and λ. The value λ specifies the maximum recursion depth, which will be discussed at the end of this subsection. The value K specifies the number of possible distances from which to select a random distance. There is a tradeoff here. Choosing larger K decreases the probability of an edge becoming broken, thereby increasing the number of edges that become satisfied. On the other hand, larger K also leads to higher I/O cost. A good value is selected in Section 4.

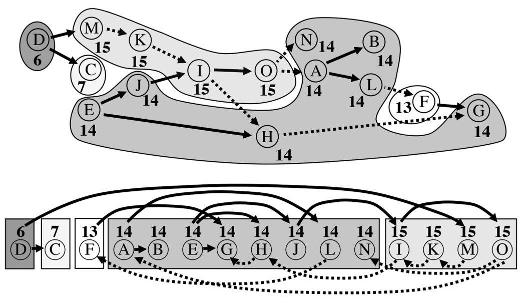
The algorithm. For now, ignore the recursion depth, λ, and the specific range of distances. The algorithm RecurTS($G, i, j, depth$) operates on the induced subgraph $G[i\,..\,j]$ as follows. Choose a distance d uniformly at random from a contiguous range of K possible distances. Assign each vertex v a distinct random priority $\rho(v)$. For each v, let $l(v)$ denote the highest priority from among v's d-hop predecessors, i.e., $l(v) = \max\{\rho(u) : u \preceq_d v \text{ in } G[i\,..\,j]\}$. Sort the vertices by $l(v)$ using



(a) Graph with priorities



(b) Labels after propagations



(c) Sorted by label, with subproblems shaded

Fig. 1. An example of a single level of recursion for Algorithm 1, with $d = 2$. Each of the three subfigures shows two equivalent images of the same graph, with the bottom image displaying the current vertex ordering from left to right. Vertices are labeled alphabetically by their initial ordering. Solid arrows represent satisfied edges (i.e., those edges directed to the right in the vertex-ordered graph) and dashed arrows represent violated edges (those edges directed to the left). The number over each vertex is either its random priority (in (a)) or its label (in (b) and (c)).

a stable sort. This is the only place in the algorithm where vertices are reordered and edges may become satisfied. At this point, vertices with the same label l are grouped together into contiguous subarrays, and the groups are sorted by label. Finally, recurse on each group.

Figure 1 illustrates an example of a single level of recursion. In the figure, the distance used is $d = 2$. The three subfigures illustrate (Figure 1(a)) the initial graph and vertex ordering, (Figure 1(b)) the labels assigned to vertices after propagating priorities to a distance of $d = 2$, and (Figure 1(c)) the new ordering on vertices and the recursive subproblems. After reordering vertices here according to label, two previously violated edges, namely, (D, C) and (J, I), become satisfied. Notice that only some of the edges crossing subproblem boundaries transition from violated to satisfied, and no edges change from satisfied to violated.

Distance ranges and maximum recursion depth. One component of the analysis (Section 4) is that the number of d-hop predecessors of v decreases with each level of recursion. This progress argument, however, is with respect to the specific distance, and it seems difficult to argue anything about the number of d'-hop predecessors for $d' > d$. On the other hand, to argue that edges are unlikely to be broken, distances need to be selected randomly from K possibilities. To reconcile these two issues, the range of distances depends on the level of recursion, decreasing with each level. Moreover, since distances should always be positive, the distance used at recursion depth 0 places a limit on the number of levels of recursion that the algorithm can support.

Putting these ideas together, we have the following. If a call is made with recursion depth $depth \geq \lambda$, then the algorithm simply returns. Otherwise, the distance d is selected uniformly at random from the range $[d_{\min}, d_{\max})$, where $d_{\min} = d_{\max} - K$ and $d_{\max} = (\lambda - depth) \cdot K$.

3.2 Achieving I/O Efficiency

This section describes how to make RecurTS I/O efficient, all of which is fairly straightforward. We first describe the implementation with respect to the initial call to RecurTS on the entire graph. We later describe how to implement the recursion.

Each vertex and edge record is augmented with a constant amount of additional information, so that the total space of the vertex and edge arrays is still $O(V)$ and $O(E)$, respectively. The standard technique for transferring information along edges is by performing sorts and scans of the vertex and edge arrays. These sorts should be viewed as transient, unlike the sort explicitly given in Algorithm 1 whose goal is to produce the topological sort.

First, tag each vertex v with its index $index(v)$, which can be achieved by a single scan (i.e.„ iterating in order) of the vertex array. Next, tag each edge by the indices of its endpoints. This edge-tagging step can be accomplished by sorting both the vertex array by vertex ID and sorting the edge array by the ID of one endpoint. Then, perform simultaneous scans of the vertex array and edge array, synchronizing the scans on ID, and copying the index of the vertex to the edges with matching endpoint ID. To store the index of the other endpoint, sort by the other endpoint. The cost of these steps is $\Theta(sort(V) + sort(E))$ for sorting the arrays and $O(scan(V) + scan(E))$ for iterating over them. The sort bound dominates.

To assign a permutation of priorities, simply select random numbers in the range $1, 2, \ldots, |V|^c$ for each vertex, where $c \geq 2$ is a constant that controls failure probability. Sort the vertices by priority and perform a scan to verify that all priorities are distinct. Repeat this process until the priorities are distinct.

Propagating priorities. The most difficult aspect is implementing the label $l(v) = \max\{\rho(u) : u \preceq_d v\}$. This is achieved incrementally through a sequence of propagation steps. Initially, set $l(v) = \rho(v)$ and perform an update called ***satisfied-edge propagation***. Next, perform d rounds, each including ***violated-edge propagation*** followed by satisfied-edge propagation. There are thus $2d + 1$ propagation steps in total.

Before describing how to implement the two types of propagation steps, let us first discuss the goal of each type of update. Let $l(v)$ and $l'(v)$ denote v's labels at the start and end, respectively, of a single propagation step (satisfied or violated). The goal of satisfied-edge propagation is to update the label to $l'(v) = \max\{l(u) : u \preceq_0 v\}$, i.e., propagate the label arbitrarily far but along satisfied edges only. The goal of violated-edge propagation is to update the label to $l'(v) = \max\{l(v), \max\{l(u) : (u, v) \in E\}\}$, i.e., propagate the label along a single hop that is allowed to be violated.

We now argue that the sequence of propagation steps gives each vertex the intended label.

Lemma 3.2. *After $d + 1$ satisfied-edge propagation steps interleaved with d violated-edge propagation steps, we have $l(v) = \max\{\rho(u) : u \preceq_d v\}$ for all $v \in V$.*

Proof. The proof is by induction on d. The base case is $d = 0$, meaning that we are considering the result of the first satisfied-edge propagation. Initially, $l(u) = \rho(u)$ for all $u \in V$. The propagation step then updates the labels to $l'(v) = \max\{l(u) : u \preceq_0 v\} = \max\{\rho(u) : u \preceq_0 v\}$, which satisfies the claim for $d = 0$.

For the inductive step, assume that $l(x) = \max\{\rho(u) : u \preceq_{d-1} x\}$ for all $x \in V$ after the first $d - 1$ rounds, and consider the effect of performing one more round consisting of a violated-edge propagation step followed by a satisfied-edge propagation step. The violated-edge propagation updates the labels to $l'(y) = \max\{l(x) : (x,y) \in E\}$. By the inductive assumption on $l(x)$, this reduces to $l'(y) = \max\{\rho(u) : u \preceq_{d-1} x, (x,y) \in E\}$. The subsequent satisfied-edge propagation step updates the labels once more to $l''(v) = \max\{l'(y) : y \preceq_0 v\}$, which when substituting in $l'(y)$ gives $l''(v) = \max\{\rho(u) : u \preceq_{d-1} x, (x,y) \in E, y \preceq_0 v\}$. Observing that any path containing d violated hops can be broken down into a path of $d-1$ violated hops followed by a single violated edge followed by a satisfied path, as in the preceding expression, completes the proof. □

Implementing satisfied-edge propagation. The satisfied edges E_{sat} can be identified by scanning through the edge set and identifying those edges (u,v) with $index(u) < index(v)$. Note that when V is sorted by index, the graph $G_{sat} = (V, E_{sat})$ is topologically sorted, which is important as we shall apply time-forward processing.

In more detail, performing the update $l'(v) = \max\{l(u) : u \preceq_0 v \text{ in } G\}$ is equivalent to computing $l'(v) = \max\{l(u) : u \preceq_\infty v \text{ in } G_{sat}\}$. The following is a simple sequential algorithm for computing the updated label with regard to G_{sat}. Consider the vertices v in G_{sat} in topological-sort order; update v's label to $\max\{l(v), \max\{l(u) : (u,v) \in E_{sat}\}\}$, i.e., the maximum of its old value and the value on all immediate predecessors. This local-update rule is exactly the kind that can be implemented I/O-efficiently using time-forward processing [4, 10], assuming G_{sat} is topologically sorted.

Time-forward processing. This section describes the algorithm for time-forward processing from [4, 10]. The algorithm takes as input a DAG with the vertices in topologically sorted order. We assume the edges are given in a list in no particular order to start. Each vertex v also has a priority $\rho(v)$. The goal is to compute $f(v)$ for each vertex where $f(v) = \max(p(v), f(u))$ for each (u,v) in E, where $f(u)$ is defined recursively. We use an external memory priority queue, which has amortized cost $O(1/B \log_{M/B} N/B)$ I/Os for insert, delete, and extract min [4].

The vertices are processed in topological order. To process a vertex v, first compute $f(v)$, which will be described later. Then for each outgoing edge from v, $(v,w) \in E$, insert (v,w) with key w, i.e., the position of w in topologically sorted order, into the priority queue augmented with $f(v)$. Notice that each edge is inserted into the priority queue once, and each vertex has an element in the priority queue for each incoming edge it has. To compute $f(v)$, we must extract min once for each incoming edge that v has. Then set $f(v)$ to be the maximum of all $f(w)$ augmented to each incoming edge (w,v), and $\rho(v)$.

The edges are inserted in topological order of the source vertex. By sorting the edge list by source vertex, this allows for one scan of the edge list to insert all the edges into the priority queue. The number of inserts to the priority queue is the number of edges $O(E)$. The number of extract mins is also the number of edges $O(E)$. In total, this is $O(E/B \log_{M/B} E/B)$ I/Os.

Implementing violated-edge propagation. This step can be accomplished by sorting and scanning. In particular, first sort the edges (u,v) by $index(u)$ so that all outgoing edges for a particular vertex u are consecutive. Then scan through the vertices and edges simultaneously, synchronizing on the vertex index. Attach to the edge (u,v) the priority $l(u,v) = l(u)$. Next, sort the edges (u,v) by $index(v)$. Now the incoming edges for each vertex v are consecutive. Finally, scan through the edges and vertices simultaneously, and for each v update $l'(v) = \max\{l(v), \max\{l(u) : (u,v) \in E\}\}$.

Implementing the recursion. To slightly simplify the analysis of the I/O cost, it is convenient to reason about the algorithm as performing the recursion level by level.[1] That is, do not actually

[1]The issue is that there are no bounds on the relative sizes of a problem and its recursive subproblems—a graph that fits in cache may be partitioned into subgraphs that are much smaller than a block. If considering each recursive subproblem one at a time, the analysis would have to be careful about the accounting of these small subproblems.

make multiple recursive calls. Instead, perform the algorithm as described for the entire level at once. The only additional bookkeeping necessary is to delimit the boundaries between each recursive subproblem in the vertex array. Each level of recursion can then be implemented by first scanning through the vertex array and tagging each vertex with a subproblem ID (increasing by one when crossing each subproblem boundary) and similarly tagging the edges with the subproblem IDs of its endpoints. All edges whose endpoints have different subproblems should be ignored in all steps. Whenever sorting vertices by label or priority, the subproblem ID should also be taken into account as the most significant feature in the sort. (That is, sort lexicographically by subproblem ID, then label/priority.) The other details are unchanged.

We can now analyze the I/O cost of the recursive algorithm. Assuming a minimum constant size on the cache is necessary to implement a constant number of synchronized scans in $O(scan(N))$ I/Os. There are also similar cache-size assumptions in time-forward processing [4] (which are not highlighted in those theorem statements) that would carry over to this setting. The following theorem comes from Theorem 3 and Section 4.1 in [4].

Theorem 3.3 (From [4]). *There exists a constant δ such that if there is a cache with at least δ blocks, then given a DAG in topologically sorted order, time-forward processing can be performed in $O(sort(E))$ I/Os.*

Lemma 3.4. *There exists a constant δ such that the following holds: if the cache contains at least δ blocks, then a single execution of RecurTS has I/O cost $O(K\lambda^2 sort(E))$, with high probability. The $K \geq 1$ and $\lambda \geq 1$ here are the global parameters of the algorithm.*

Proof. Each of the λ levels of recursion performs d rounds of label propagation, where d is at most λK. Each round of label propagation can be implemented in $O(sort(E))$ I/Os. All other steps of the algorithm can also be accomplished in a constant number of scans and sorts, except assigning priorities performs a single attempt with high probability. (The failure probability depends on the range of priorities.) The cost of each level of the λ levels of recursion is thus $O(K\lambda sort(E))$. □

4 TOPOLOGICAL SORT ANALYSIS

The section analyzes the topological-sort algorithm given in Section 3. The goal is to show that, with high probability, the main algorithm completes after $O(\log V)$ executions of RecurTS. The key component toward achieving this goal is to show that in each execution, each violated edge has a constant probability of becoming satisfied. The bulk of this section is devoted to proving this claim. Given the claim, it is simple to show that $O(\log V)$ executions suffice.

Consider any violated edge (u, v) and an execution of RecurTS. The most important point of the execution is the moment, if any, that u and v receive different priorities and are hence placed in different recursive subproblems. This step is the only time during the execution that the relative order of u and v may change. If u is ordered before v, then the edge becomes satisfied. If u remains ordered after v, however, then the edge (u, v) cannot become satisfied for the remainder of the execution (i.e., until the next execution of RecurTS.) The following definition captures this bad outcome:

Definition 4.1. An edge (u, v) is ***broken*** if (i) $index(v) < index(u)$, i.e., v precedes u in the ordering of vertices, and (ii) u and v are in different recursive subproblems.

In Figure 1(c), the broken edges are (L, F), (I, H), (O, A), and (O, N).

As outlined in Section 1.3.1, our analysis consists of two main components. First, we argue that, for large enough K, an edge (u, v) has at most constant probability of becoming broken during an execution of RecurTS. Second, we argue that for large enough λ, the execution is likely to terminate with u and v in different subproblems. If u and v are in different subproblems, and the edge is not broken, then it must be satisfied. The remainder of the section focuses on proving each of these claims.

Predecessors. Throughout the analysis, it is useful to refer to the set of predecessors of a particular vertex. Let $G = (V, E)$ be a graph, let $v \in V$ be a vertex in the graph, and let d be a distance. We define the ***d*-hop predecessors of** v, denoted by $A(G, v, d)$, as $A(G, v, d) = \{x : x \preceq_d v \text{ in } G\}$.

4.1 Bounding the Probability of an Edge Becoming Broken

We argue that in any level of recursion, a particular violated edge (u, v) has probability at most $O(\log V/K)$ of becoming broken. Taking a union bound across all λ levels of recursion gives a probability of at most $O(\lambda \log V/K)$ that the edge becomes broken across the entire execution of RecurTS. Setting $K = \Omega(\lambda \log V)$ and tuning constants appropriately, the probability that (u, v) becomes broken is upper bounded by a constant.

We begin by considering how an edge can become broken. The following lemma implies that an edge (u, v) can become broken only if u's highest-priority d-hop predecessor is located exactly d violated hops away.

Lemma 4.2. *Consider a call RecurTS$(G, i, j, depth)$. Let $G' = G[i\,..\,j]$ denote the induced subgraph, let (u, v) be an edge in G', and let d be the random distance chosen. Finally, let x denote the vertex in $A(G', u, d)$ with the highest priority $\rho(x)$. If $x \in A(G', u, d-1)$, then $l(u) \leq l(v)$.*

Proof. Suppose $x \in A(G', u, d-1)$. Then we have $x \preceq_{d-1} u$ and $u \preceq_1 v$, giving $x \preceq_d v$. It follows that $l(v) \geq \rho(x)$. Moreover, since x is the vertex with highest priority among u's d-hop predecessors, $l(u) = \rho(x)$. We thus have $l(v) \geq \rho(x) = l(u)$. □

We next bound the probability that an edge (u, v) becomes broken in a particular recursive call. Consider the random process as follows. First choose a random distance. Then identify which vertex, from among the d-hop predecessors, has highest priority. Specifically, determine if the highest-priority predecessor is also a $(d-1)$-hop predecessor; if so, by the previous lemma the edge does not break. The probability of the edge breaking thus depends on the relative sizes of the $A(G', u, d)$ and $A(G', u, d-1)$. The main idea is therefore to characterize distances by relative neighborhood sizes.

The argument is roughly as follows, but the following lemma provides a tighter bound. A distance d is "bad" if at least a $1/\log V$-fraction of the d-hop predecessors are at distance exactly d, i.e., not also $(d-1)$-hop predecessors. If a bad distance is selected, the probability of the edge breaking may be high. Fortunately, due to the expansion implied by bad distances, there cannot be too many bad distances—specifically only $O(\log^2 V)$ of them. If a good distance is selected, the probability that the edge breaks is at most $O(1/\log V)$. Putting these together, the probability that the edge breaks is $O(\log^2 V/K + 1/\log V)$. The next lemma improves this to $O(\log V/K)$ by more carefully accounting for how bad each distance is.

Lemma 4.3. *Consider a call RecurTS$(G, i, j, depth)$. Let $G' = G[i\,..\,j]$ denote the induced subgraph and let (u, v) be a violated edge in G'. Then the probability that the edge becomes broken during this call is at most* $\lg(|V|)/K$.

Proof. Let B denote the event that the edge (u, v) is broken. Let d denote the random distance chosen, and let x be the vertex in $A(G', u, d)$ with highest priority. By Lemma 4.2, $\mathbf{Pr}[\,B\,] \leq \mathbf{Pr}[\,x \notin A(G', u, d-1)\,]$, so it suffices to bound the latter.

For each possible d in $[d_{\min} - 1, d_{\max})$, let $s_d = |A(G', u, d)|$ denote the number of d-hop predecessors of u in G'. Define $\gamma_d = s_{d-1}/s_d$ to be the fraction of of u's d-hop predecessors that are also $(d-1)$-hop predecessors.

Let E_d denote the event that distance d is chosen. Once d is fixed, we trivially have $\mathbf{Pr}\left[\, B|E_d \,\right] \leq 1 - \gamma_d$. Since the distance is chosen uniformly at random from K possibilities, we have

$$\mathbf{Pr}\left[\, B \,\right] = \sum_{d=d_{\min}}^{d_{\max}-1} (\mathbf{Pr}\left[\, B|E_d \,\right] \cdot \mathbf{Pr}\left[\, E_d \,\right]) \leq \sum_{d=d_{\min}}^{d_{\max}-1} (1-\gamma_d)/K.$$

The vertex u is a d-hop predecessor of itself, and at most every vertex is a d-hop predecessor of u, so $1 \leq s_d \leq |V|$. We therefore have $|V| \geq s_{d_{\max}-1} \geq s_{d_{\max}-1}/s_{d_{\min}-1} = \prod_{d=d_{\min}}^{d_{\max}-1}(1/\gamma_d)$. By monotonicity of the lg function, $\lg(|V|) \geq \sum_{d=d_{\min}}^{d_{\max}-1} \lg(1/\gamma_d)$. Finally, $\gamma_d \in (0, 1]$, and for this range $\lg(1/\gamma_d) \geq 1 - \gamma_d$. We therefore have $\lg(|V|) \geq \sum_{d=d_{\min}}^{d_{\max}-1}(1-\gamma_d)$.

Substituting back for the probability of B, we have

$$\mathbf{Pr}\left[\, B \,\right] \leq \frac{1}{K} \sum_{d=d_{\min}}^{d_{\max}-1} (1-\gamma_d) \leq \frac{\lg(|V|)}{K}. \qquad \square$$

4.2 Bounding the Probability that An Edge Crosses Subproblems

The second key component of the analysis is to argue that at the end of an execution, the edge (u, v) is likely to cross subproblem boundaries. To achieve this goal, we argue that with each level of recursion, v is likely to lose a constant fraction of its nearby predecessors. Thus, with $\Omega(\log V)$ levels of recursion, it is very likely that v has no predecessors. If v has no predecessors, then u must be in a different subproblem.

Definitions. More formally, consider the calls RecurTS(G, i, j, ℓ) arising during the execution of the recursive algorithm, where ℓ here denotes level or depth of the recursion. If $v \in G\left[i\,..\,j\right]$, we call $G_v^\ell = G\left[i\,..\,j\right]$ the ***level-ℓ graph of*** v. Notice that v belongs to at most one subproblem at each level of recursion. If v does not belong to any level-ℓ subproblems (i.e., if the base case was reached early), then G_v^ℓ is the empty graph. Thus, v has a corresponding sequence $G_v^0, G_v^1, \ldots, G_v^\lambda$ of level-$0, 1, \ldots, \lambda$ graphs, where $G_v^0 \supseteq G_v^1 \supseteq \cdots \supseteq G_v^\lambda$.

For this subsection, the important feature is the number of nearby, proper predecessors v has at each level of recursion. A vertex x is a ***level-ℓ active predecessor*** of v if $x \neq v$ and $x \preceq_{d_{\max}} v$ in G_v^ℓ, where $d_{\max} = K(\lambda - \ell)$ is the maximum distance for this level of recursion. Notice that v is not an active predecessor of itself.

Reducing the number of active predecessors. We start with a simple observation, captured by the first lemma: no new relationships are created between vertices as the algorithm recurses. Thus, we need not worry about the set of active predecessors growing—the only challenge is to show that a significant fraction of the predecessors are likely to be knocked out.

Lemma 4.4. *Consider any vertex v and its level-$(\ell - 1)$ and level-(ℓ) subgraphs $G_v^{\ell-1}$ and G_v^ℓ, respectively. For any vertex x and distance d, if $x \npreceq_d v$ in $G_v^{\ell-1}$, then $x \npreceq_d v$ in G_v^ℓ.*

Proof. At first glance, this statement sounds obvious given that edges are never created. There is, however, one concern—satisfying edges can decrease distances between vertices. Since the relative order of vertices only changes when those vertices are placed in different recursive subproblems, an edge can only be satisfied in G_v^ℓ if it is already satisfied in $G_v^{\ell-1}$. $\square$

We are now ready to argue that the number of active predecessors is likely to decrease at each level of recursion. This proof leverages only the random priorities—the fact that distances are chosen randomly is not important. The proof (notably the second claim therein) lifts some ideas from [16, Lemma 3.4].

Lemma 4.5. *Consider any vertex v and level ℓ of recursion. Let α and α' denote the number of level-ℓ and level-$(\ell+1)$, respectively, active predecessors of v. Then* $\mathbf{Pr}\left[\, \alpha' \leq (3/4)\alpha \,\right] \geq 1/3$.

Proof. If $\alpha = 0$, the claim is trivial. Otherwise, consider the level-ℓ call to RecurTS on graph G_v^ℓ. Let d be the distance selected (which need not be random for the purpose of the proof). Let $A_d = A(G_v^\ell, v, d) \setminus \{v\}$ denote the set of d-hop predecessors of v in G_v^ℓ, excluding v itself, and let $a_d = |A_d|$. Notice by Lemma 4.4 and the decreasing distance ranges, the level-$(\ell+1)$ active predecessors are a subset of A_d. Moreover, A_d is a subset of the level-ℓ active predecessors. It is thus sufficient to argue that with probability at least 1/3, at most $3a_d/4 \leq 3\alpha/4$ of the vertices in A_d are also in $G_v^{\ell+1}$.

Let x be a random variable denoting the highest-priority vertex in $A_d \cup \{v\}$. For any other vertex $y \in A_d$, we say that x ***knocks out*** y if $x \npreceq_d y$. The remainder of the proof amounts to proving the following two claims: (1) If x knocks out y, then $y \notin G_v^{\ell+1}$, and (2) with probability at least 1/3, x knocks out at least $a_d/4$ vertices from A_d.

Claim 1. Recall that x is the highest-priority vertex from $A_d \cup \{v\}$. Thus, v inherits the label $l(v) = \rho(x)$, which defines its subproblem. If $x \npreceq_d y$, then $l(y) \neq \rho(x)$, and y is in a different subproblem from v.

Claim 2. Because the graph is acyclic, for any pair $x \neq y$ of vertices, at least one of the following must be true: $x \npreceq_d y$ or $y \npreceq_d x$. Moreover, for all $y \in A_d$, $y \preceq_d v$ by definition, so $v \npreceq_d y$. Thus, the total number of pairs $x \in A_d \cup \{v\}$ and $y \neq x \in A_d$ for which x knocks out y must be at least $\binom{a_d}{2} + a_d = a_d(a_d+1)/2$. Because x is selected from $a_d + 1$ choices, the expected number of vertices knocked out by x is at least $a_d/2$. By Markov's inequality, the probability that at least $(3/4)a_d$ vertices are *not* knocked out is therefore at most 2/3. □

Lemma 4.5 indicates that with each level of recursion, the number of active predecessors is likely to decrease by a constant factor. The following lemma says that after enough levels of recursion, v is likely to have no remaining active predecessors. The implication is that all of its incoming edges cross subproblem boundaries.

Lemma 4.6. *Consider any vertex v and a complete execution of the recursive algorithm. With probability at least $1 - 8\lg|V|/\lambda$, v has no active predecessors at the λ-th level of recursion.*

Proof. Let X_j be a random variable denoting the number of levels ℓ for which the number of active predecessors of v falls in the range $[(3/4)^{j+1}, (3/4)^j) \cdot |V|$, for integer j. When $j \geq \log_{4/3}|V|$, the high end of the range is strictly less than $1/|V| \cdot |V| = 1$, meaning that v has no active predecessors remaining. Let $X = \sum_{j<\log_{4/3}|V|} X_j$. If $X < \lambda$, then at or before the λ-th level of recursion, v has no active predecessors. Our goal is thus to argue that this event is likely to occur.

Lemma 4.5 implies that the number of rounds necessary to get a (3/4) reduction is at most three in expectation. Thus, $E[X_j] \leq E[X_j | X_j \geq 1] \leq 3$. By linearity of expectation, $E[X] \leq 3\log_{4/3}(|V|) < 8\lg(|V|)$. By Markov's inequality, $\mathbf{Pr}\left[\, X \geq \lambda \,\right] \leq 8\lg(|V|)/(\lambda)$. □

4.3 Edges are Likely to Become Satisfied

Thus far, we have argued that edges are unlikely to become broken at any particular level of recursion, and that edges are likely to cross subproblem boundaries by the time the recursive algorithm terminates. This section combines those pieces to conclude that in a single execution of the recursive algorithm, a violated edge is likely to become satisfied.

Before getting to the main claim, we first observe that satisfied edges stay satisfied. This fact is important both to argue that a violated edge is likely to become satisfied in a single execution, and to argue that multiple executions lead to monotonic progress.

Lemma 4.7. *Consider an execution of the recursive algorithm RecurTS. If an edge (u, v) is satisfied at the ℓ-th level of recursion, then it is satisfied at all subsequent levels of recursion.*

Proof. Proof by induction on the level of recursion. Consider the call at the ℓth level of recursion, and suppose that the edge (u, v) is satisfied at the start of the call. The goal is to show that it remains satisfied in the next recursive subproblem. Note that if (u, v) is satisfied at the start of the call, then $u \preceq_0 v$ in the current graph. Let $l(u)$ be the final label on vertex u, and let x be the vertex such that $l(u) = \rho(x)$. Then $x \preceq_d u$, which coupled with $u \preceq_0 v$ implies that $x \preceq_d v$. It follows that $l(v) \geq l(u)$. If $l(v) = l(u)$, then u and v maintain their current ordering. If $l(v) > l(u)$, then v is placed in an even later subproblem. Either way, (u, v) remains satisfied. □

Lemma 4.8. *Let (u, v) be any edge, and consider a complete execution of RecurTS with parameters $\lambda \geq 32 \lg V$ and $K \geq 4\lambda \lg V$. If (u, v) is violated initially, then with probability at least $1/2$, (u, v) is satisfied at the end of the execution. If (u, v) is satisfied initially, then with probability 1 it is still satisfied at the end.*

Proof. By Lemma 4.7, a satisfied edge always remains satisfied. The remainder focuses on the case that (u, v) is initially violated.

Let A be the event that (u, v) is violated at the end of the execution. Let B be the event that the edge breaks at some level of recursion, and let C be the event that the two endpoints u and v are in the same level-λ subproblem. If neither B nor C occurs, then the edge crosses properly ordered subproblems and the edge is satisfied. We thus have $\mathbf{Pr}[A] \leq \mathbf{Pr}[B] + \mathbf{Pr}[C]$ by a union bound.

By Lemma 4.6, $\mathbf{Pr}[C] \leq 8 \lg V/\lambda \leq 1/4$ for the specified choice of λ. By Lemma 4.3, the probability of breaking at any individual level is at most $\lg V/K$. Taking a union bound across λ levels, we have $\mathbf{Pr}[B] \leq \lambda \lg V/K \leq 1/4$ for the specified choice of K. Adding these together gives total failure probability of at most $1/2$. □

4.4 Bounds on the Main Algorithm

Finally, we analyze the main algorithm, which repeatedly executes RecurTS until the graph is topologically sorted.

Theorem 4.9. *Let $G = (V, E)$ be any directed acyclic graph, and choose $\lambda \geq 32 \lg V$ and $K \geq 4\lambda \lg V$. Then for any $c > 0$, with failure probability at most $1/|V|^c$, the graph is topologically sorted after at most $\lceil (c+2) \lg V \rceil$ iterations of the recursive algorithm.*

Proof. Consider any initially violated edge (u, v). By Lemma 4.8, each execution of RecurTS satisfies the edge with probability at least $1/2$. Moreover, if the edge is satisfied in any iteration, it remains satisfied for all subsequent iterations. The probability that the edge is still violated after $\lceil (c+2) \lg V \rceil$ iterations is therefore at most $(1/2)^{(c+2)\lg V} = 1/|V|^{c+2}$. Taking a union bound across less than $|V|^2$ possible edges completes the proof. □

Theorem 4.10. *For any directed acyclic graph $G = (V, E)$, there exist settings of K and λ such that the algorithm topologically sorts the graph in $O(sort(E) \cdot \log^5 V)$ I/Os, with high probability.*

Proof. From Lemma 3.4, a single execution of RecurTS has I/O cost $O(K\lambda^2 \cdot sort(E))$, with high probability. Theorem 4.9 states that $O(\log V)$ executions suffice, with high probability, for

ALGORITHM 2: Conceptual Algorithm for Strongly Connected Components

```
1: function SCC($G = (V, E)$)
2:     $H = (V_H, E_H)$, where $V_H = V$ and $E_H = E$ initially
3:     repeat until $H$ is topologically sorted
4:         RecurSCC($H, 1, |V_H|, 0$)
5:         Perform contraction step and update $H$

6: function RecurSCC($H, i, j, depth$)          ▷ Reorders the subarray $V_H[i\,..\,j]$ of vertices
7:     $H' = H[i\,..\,j]$
8:     if $depth \geq \lambda$ or $i = j$ then return
9:     $d_{\max} = (\lambda - depth) \cdot K$ and $d_{\min} = d_{\max} - K$
10:    Choose $d$ uniformly at random from $[d_{\min}, d_{\max})$
11:    Choose a uniformly random permutation of priorities $\{\rho(v)\}$
12:    For all $v$, compute $f(v) = \max\{\rho(u) : u \preceq_d v \text{ in } H'\}$
13:    For all $v$, compute $b(v) = \max\{\rho(w) : v \preceq_d w \text{ in } H'\}$
14:    for each $v \in V_H[i\,..\,j]$
15:        $l(v) = \begin{cases} -b(v) & \text{if } b(v) > f(v) \text{ (backward search dominates)} \\ -b(v) + 1/2 & \text{if } b(v) = f(v) \text{ (strongly connected)} \\ f(v) & \text{if } b(v) < f(v) \text{ (forward search dominates)} \end{cases}$
16:    Sort vertices $V_H[i\,..\,j]$ lexicographically by $\langle l(v), index(v) \rangle$
17:    Partition $V_H[i\,..\,j]$ into maximal groups $[i_1, j_1], [i_2, j_2], \ldots, [i_t, j_t]$ of a single label ($i_r = j_{r-1} + 1$)
18:    for $r = 1$ to $t$
19:        if $f(V_H[i_r]) = b(V_H[i_r])$ then          ▷ do not recurse on strongly connected groups
20:            mark the vertices in the group for contraction
21:        else RecurSCC($H, i_r, j_r, depth + 1$)
```

$\lambda = \Theta(\log V)$ and $K = \Theta(\lambda \log V) = \Theta(\log^2 V)$. Multiplying the $O(\log V)$ executions by the cost per execution gives the theorem. □

5 STRONGLY CONNECTED COMPONENTS

This section describes our algorithm for strongly connected components. Given a graph $G = (V, E)$, vertices $u, v \in V$ are ***strongly connected*** if there exist directed paths both from u to v and from v to u. A ***strongly connected component*** is a maximal set of vertices such that every pair of vertices therein is strongly connected. The ***condensation*** H of a graph G is the DAG of strongly connected components, i.e., the graph formed if each strongly connected component is contracted. The goal is to identify for each vertex the strongly connected component to which it belongs and to topologically sort the condensation.

At a high level, the main intent of the algorithm is similar to Algorithm 1—reorder vertices to satisfy more edges. But it would, of course, be impossible to simultaneously satisfy all edges on a cycle. Our algorithm for strongly connected components therefore performs a little extra work to identify strongly connected vertices, notably those falling on short cycles, and contracts them into a single supervertex. The graph is thus gradually transformed into its condensation; with each iteration, the number of violated edges may reduce both by removing contracted edges from the graph and by reordering any remaining supervertices.

Aside from the contraction, component maintenance, and extra bookkeeping, the main difference between the algorithms for topological sort and strongly connected components is that the former propagates priorities in only the forward direction, whereas the latter propagates priorities both forwards and backwards. This two-directional propagation facilitates the discovery of cycles.

5.1 Algorithm

Algorithm 2 presents a conceptual version of the algorithm for topologically sorting the condensation H of the graph $G = (V, E)$. Section 5.2 provides implementation details for mapping this algorithm to the I/O model.

The algorithm maintains three types of information: (1) a mapping from vertices in the original graph to (partial) components, where each partial component is a subset of vertices in a strongly connected component; (2) a graph $H = (V_H, E_H)$ on the partial components, corresponding to the graph formed by contracting each partial component in G; and (3) an ordering of the vertices V_H in the component graph. As the algorithm progresses, components are merged together through contraction steps. When the algorithm terminates, H is the condensation, and the vertex ordering represents a topological sort of the condensation.

As before, the top-level algorithm consists of multiple iterations. But now each iteration consists of not only an execution of RecurSCC, but also a contraction step following the execution. Each execution of RecurSCC is analogous to RecurTS, except that some vertices are flagged for contraction. Specifically, the output of RecurSCC is an updated ordering of the vertices V_H in the component graph H as before, but unlike RecurTS some contiguous sets of vertices are flagged for contraction. Any initially satisfied edges between unflagged vertices remain satisfied, as before, and ideally some violated edges become satisfied. During the contraction step, sets of vertices identified as being strongly connected are contracted, removing any edges between them.

The Recursive Subroutine

The recursive subroutine RecurSCC is parameterized by global values λ and K, denoting the maximum recursion depth and range of distances to choose from, respectively. RecurSCC takes as input an induced subgraph $H[i \,..\, j]$ of the graph on partial components, and the current recursion depth *depth*.

RecurSCC proceeds as follows. Much of the algorithm is similar to RecurTS of Section 3. Firstly, check if the recursion depth is exceeded (i.e., $depth \geq \lambda$), and if so simply return. Otherwise, choose a distance d uniformly at random from the range $[d_{\min}, d_{\max})$, where $d_{\min} = d_{\max} - K$. As in Section 3, the offset for the range is chosen according to the recursion depth, with $d_{\max} = (\lambda - depth) \cdot K$.

Next, assign a uniformly random permutation of priorities $\rho(v)$ to each vertex. Unlike RecurTS, RecurSCC propagates the priorities in both the forward direction and the backward direction. Specifically, define $f(v) = \max\{\rho(u) : u \preceq_d v\}$ and $b(v) = \max\{\rho(w) : v \preceq_d w\}$. Assign a label $l(v)$ to each vertex v based on the results of the forward and backward searches. There are three cases. If the priority from the forward search dominates, i.e., $f(v) > b(v)$, then $l(v) = f(v)$ as in RecurTS. If the priority from the backward search dominates, i.e., $b(v) > f(v)$, then $l(v) = -b(v)$. These two cases are symmetric—vertices dominated by larger priorities in the forward direction are pushed later in the ordering, and vertices dominated by larger priorities in the backward direction are pushed earlier in the ordering. The third case is if the priorities are equal in both directions. In this case, set $l(v) = -b(v) + 1/2$.

Finally, sort the vertices by $l(v)$, with ties broken according to the current ordering. After sorting, partition the vertices into groups of vertices having the same label, as in RecurTS. Recurse on those groups with integer labels, i.e., $f(v) \neq b(v)$. The groups with non-integer labels are instead flagged for contraction.

Note that the specific choice of label $-b(v) + 1/2$ for the third case is not particularly important. The only truly important aspect is that $-b(v) < l(v) < f(v)$ to ensure that satisfied edges remain satisfied. In fact, the ordering across groups of vertices that fall in this third case, for different priorities, does not matter. It is, however, easier to implement the subsequent contraction if each such group of vertices is contiguous. To achieve that, we include the dominating priority in the label, e.g., choosing $l(v) = -b(v) + 1/2$; many other choices would also suffice.

The main theorem, proved in Section 5.4, is the following.

Theorem 5.1. *Let $G = (V, E)$ be any directed graph. There exist settings of constants c_1 and c_2 such that for $\lambda \geq c_1 \lg V$ and $K \geq c_2 \lambda \lg V$, the following holds. For any $c > 0$, with failure probability at most $1/V^c$, the algorithm terminates within $\lceil (c+2) \lg V \rceil$ iterations of the main loop.*

5.2 I/O-Efficient Details

This section provides details on making the algorithm I/O efficient. The original vertices of graph G are stored in an array V. A second array V_H stores the vertices of graph H. Each vertex in H corresponds to a partial component in G that has been contracted, identified by the ID of a representative vertex. The edges between components are stored in an array E_H, with each edge storing the component IDs of its endpoints.

All vertex records $u \in V$ for the original graph are tagged with the ID $c(u)$ of their component's representative, which corresponds to the ID of a vertex in V_H. For convenience, the vertex records $v \in V_H$ are also tagged with $c(v)$. Initially, $c(u) = u$ for all $u \in V$, and $V_H = V$. In general, between iterations, $c(v) = v$ if and only if $v \in V_H$. When the algorithm terminates, the vertices representing each strongly connected component are topologically ordered in V_H, and for each vertex $u \in V$, $c(u)$ specifies the representative of u's strongly connected component.

The details for the recursive algorithm are similar to those for RecurTS in Section 3.2. There are minor differences in that the priorities must be propagated in two directions, now computing $b(v)$ in addition to the $f(v)$ already computed in RecurTS. But steps for $b(v)$ are symmetric, i.e., operating on the transpose graph, which can be computed in $O(sort(E))$ I/Os.

The only significant difference is implementing the main loop, namely, in the contraction step.

Contraction. By design, when RecurSCC returns, groups of vertices that are to be contracted are contiguous in the vertex array V_H. When a group is flagged for contraction, the boundaries of the group should also be marked.

The first step of the contraction is to update $c(v)$ for all vertices $v \in V_H$ to be contracted. Specifically, the first vertex x (in array order) in each group is the representative for the group. All other vertices in the group update $c(u) = x$. This step can be accomplished by a scan of the array V_H.

The next step is to update the components for vertices stored in V. Specifically, each vertex $u \in V$ has some component ID $c(u) = v$, where $v \in V_H$. The goal is to update $c(u) = c(v)$. To do so, sort V by component IDs $c(u)$ and sort V_H by vertex IDs v. Thus, both V and V_H are sorted according to their original components. Moreover, the vertices $v \in V_H$ already know their new component $c(v)$. Next, perform synchronized scans of V and V_H, and for each vertex $u \in V$, update $c(u) = c(v)$.

Any vertices in V with $c(v) \neq v$ can now be removed from V_H. This step can be accomplished with a scan. At this point, all vertices have the correct component IDs, and only representatives are stored in V_H.

The final step is to update the edges E_H. Specifically, any edge (u, v) should be updated to reflect the component IDs of its endpoints, i.e., $(c(u), c(v))$. To do so, sort V by ID, and sort the edges $(u, v) \in E_H$ by the ID of u. Next, perform synchronized scans of E_H and V, updating u to $c(u)$ for each edge (u, v). Then sort the edges by the ID of the other endpoint v and perform a similar update. Finally, self-loops can be removed by scanning through all the edges one last time and checking for any edges of the form (u, u). Optionally, duplicate edges can also be removed by sorting the edges one last time (by both endpoints) and scanning through to remove duplicates.

5.3 I/O Complexity of Strongly Connected Components

Assuming Theorem 5.1, we now bound the I/O cost of the algorithm.

Lemma 5.2. *A single execution of* RecurSCC *has I/O cost $O(K\lambda^2 sort(E))$ I/Os, with high probability.*

Proof. Proof is identical to the proof of Lemma 3.4, except that the specific constants change because priorities must be propagated in two directions. □

Theorem 5.3. *For any directed graph $G = (V, E)$, there exist settings of K and λ such that the algorithm topologically sorts the condensation of the graph in $O(sort(E) \cdot \log^5 V)$ I/Os, with high probability.*

Proof. Theorem 5.1 states that for $\lambda = \Theta(\log V)$ and $K = \Theta(\lambda^2 \log V)$, after $O(\log V)$ executions of RecurSCC, the algorithm is successful with high probability. From Lemma 5.2, the cost of each execution of RecurSCC is $O(K\lambda^2 sort(E))$. The cost of each contraction step is $O(sort(E))$, which is dominated by the cost of executing the recursive algorithm. Therefore the total I/O cost of the full algorithm is $O(sort(E) \cdot \log^5 V)$. □

5.4 Strongly Connected Components Analysis

The goal of this section is to prove Theorem 5.1, i.e., that $O(\log V)$ executions of the main loop suffice, with high probability. The analysis follows a similar structure to the analysis of topological sort in Section 4. The main goal is to show that any violated edge $(u, v) \in E_H$ has a constant probability of either becoming satisfied in an execution of RecurSCC or being contracted away thereafter. Since any cycle in the graph must have at least one violated edge, satisfying all remaining edges implies that all cycles have been contracted, and the condensation of the graph is topologically sorted. Given that claim, it is easy to show that $O(\log V)$ iterations suffice.

As before, the analysis consists of two main components applied to graph H. A minor difference is that groups of vertices to be contracted are technically not part of a recursive subproblem. Insofar as definitions are concerned (e.g., being broken), when we say "subproblem" we mean each group of vertices produced by the partitioning step in the algorithm, either corresponding to a group to be contracted or a recursive call.

The first component of the analysis is to show that an edge is unlikely to break. This component is largely similar to the corresponding component in Section 4, except that edges may break due to searches in either direction. Note that, conveniently, edges within a group to be contracted are never broken as these edges do not cross subproblem boundaries.

The goal of the second component is now to show that for any edge $(u, v) \in E_H$, the execution of RecurSCC is likely to end either with u and v in different subproblems, or with u and v marked for contraction with each other. If the edge is not broken, and u and v are in different subproblems, then the edge becomes satisfied. If u and v are contracted, then the edge is removed from the graph entirely because the contraction step removes self-loops.

Successors. The main differences in the analysis arise from the fact that priorities are propagated in two directions. It is thus no longer sufficient to focus just on the d-hop predecessors. We must also consider the successors. We define the ***d-hop successors of*** v, denoted by $D(G, v, d)$, as $D(G, v, d) = \{x : v \preceq_d x \text{ in } G\}$.

The presence of backward propagation impacts that analysis in various places, most of which are minor. The most difficult is in arguing progress with respect to the number of nearby vertices.

Rather than argue progress on just the number of nearby (active) predecessors, Lemma 5.8 argues progress on both nearby predecessors and successors.

5.4.1 Bounding the Probability of an Edge Becoming Broken. The following lemmas are analogous to Lemmas 4.2 and 4.3. Notably, an edge cannot be broken unless either its d-hop predecessor or successor is exactly d hops away, which is unlikely to occur.

LEMMA 5.4. *Consider call* RECURSCC$(H, i, j, depth)$. *Let* $H' = H[i\,..\,j]$ *denote the induced subgraph, let* (u, v) *be an edge in* G', *and let* d *be the random distance chosen. Let* x *denote the vertex with highest priority in the set* $A(H', u, d)$. *If* $x \in A(H', u, d-1)$, *then* $f(u) \le f(v)$. *Similarly, let* y *denote the vertex with highest priority in the set* $D(H', v, d)$. *If* $y \in D(H', v, d-1)$, *then* $b(v) \le b(u)$.

PROOF. Suppose $x \in A(H', u, d-1)$. Then we have $x \preceq_{d-1} u$ and $u \preceq_1 v$, giving $x \preceq_d v$. It follows that $f(v) \ge \rho(x)$. Moreover, since x has the highest priority among u's d-hop predecessors, $f(u) = \rho(x)$. We thus have $f(v) \ge \rho(x) = f(u)$.

Suppose $y \in D(H', v, d-1)$. Then we have $u \preceq_1 v$ and $v \preceq_{d-1} y$, giving $u \preceq_d y$. It follows that $b(u) \ge \rho(y)$. Moreover, since y has the highest priority among v's d-hop predecessors, $b(v) = \rho(y)$. We thus have $b(u) \ge \rho(y) = b(v)$. □

LEMMA 5.5. *Consider call* RECURSCC$(H, i, j, depth)$. *Let* $H' = G[i\,..\,j]$ *denote the induced subgraph and let* (u, v) *be a violated edge in* H'. *Then, the probability that the edge becomes broken during this call is at most* $2\lg(|V|)/K$.

PROOF. The violated edge (u, v) becomes broken if and only if $l(u) > l(v)$, in which case v is ordered before u, and u and v are placed into different subproblems. We have $l(u) > l(v)$ only if $f(u) > f(v)$ or $b(v) > b(u)$. The probability that $f(u) > f(v)$ is the same as the proof of Lemma 4.3, but this time using Lemma 5.4. The case that $b(v) > b(u)$ is symmetric, which gives the total probability of (u, v) becoming broken during this call to be $2\lg(|V|)/K$. □

5.4.2 Bounding the Probability that An Edge Crosses Subproblems. The analysis here is analogous to Section 4.2. The main difference is that here we consider the number of active vertices in both directions, not just predecessors. For this section, we adopt the same notion of level-ℓ graphs as in Section 4.2, except applied to the contracted graph H instead of the original graph G. Note that H_v^ℓ is the empty graph if v is no longer part of a recursive subproblem, which can now also occur if v is marked for contraction before the ℓ-th level of recursion.

This first lemma says that vertices do not get closer together, i.e., no new relationships are created, when recursing. Consequently, it is sufficient to argue that a constant fraction of related vertices are likely to be knocked out.

LEMMA 5.6. *Consider any vertex* v *and its level-*$(\ell - 1)$ *and level-*(ℓ) *subgraphs* $H_v^{\ell-1}$ *and* H_v^ℓ, *respectively. For any vertex* x *and distance* d, *if* $x \npreceq_d v$ *in* $H_v^{\ell-1}$, *then* $x \npreceq_d v$ *in* H_v^ℓ. *Similarly, if* $v \npreceq_d x$ *in* $H_v^{\ell-1}$, *then* $v \npreceq_d x$ *in* H_v^ℓ.

PROOF. The distances in the graph can only decrease if edges become satisfied, new edges are incorporated, or if vertices are contracted. None of these occurs within the scope of recursive subproblems—the relative ordering within each subproblem is unchanged, and contraction only occurs between iterations. □

The remainder of the section focuses on the number of active vertices, except we now consider both predecessors and successors.

Definition 5.7. A vertex x is a ***level-ℓ active successor*** of v if $x \neq v$ and $v \preceq_{d_{\max}} x$ in H_v^ℓ, where $d_{\max} = K(\lambda - \ell)$ is the maximum distance for this level of recursion. Notice this definition is symmetric to the level-ℓ active predecessors of v and similarly, v is not an active successor of itself.

Consider any edge (u, v) that is violated at the start of the recursive algorithm. Observe that if v has no level-ℓ active predecessors, then either (u, v) falls within a group marked for contraction, or (u, v) crosses a subproblem boundary.

The analysis differs here from topological sort because the subproblem derives from both predecessors and successors. In particular, the label $l(v)$, for a vertex v, is based on whether the forwards or backwards search dominates. Our claim here is that the sum of the number of active predecessors and active successors decreases by a constant factor in each level of recursion with constant probability.

Lemma 5.8. *Consider any vertex v and level ℓ of recursion. Let A and A' be the set of level-ℓ and level-$(\ell + 1)$, respectively, active predecessors of v. Let D and D' be the set of level-ℓ and level-$(\ell + 1)$, respectively, active successors of v. Let $\eta = |A \cup D|$, and $\eta' = |A' \cup D'|$. Then* $\mathbf{Pr}\left[\, \eta' \leq (7/8)\eta \,\right] \geq 1/6$.

Proof. Consider the level-ℓ call to RecurSCC on graph H_v^ℓ. Let d be the distance selected. Let $A_d = A(H_v^\ell, v, d) \setminus \{v\}$ be the set of d-hop predecessors of v in H_v^ℓ, excluding v itself, and let $a_d = |A_d|$. Similarly, let $D_d = D(H_v^\ell, v, d) \setminus \{v\}$ be v's d-hop successors. Let $r_d = |A_d \cup D_d|$ be the total number of d-hop related vertices, in both directions, excluding v. For the remainder of the proof, assume without loss of generality (by symmetry) that $|A_d| \geq |D_d|$, and hence $a_d \geq r_d/2$.

Since $d \leq d_{\max}$, we have $A_d \subseteq A$ and $D_d \subseteq D$ and hence $r_d \leq \eta$. By Lemma 5.6 and the fact that distances decrease with each level of recursion, $A' \subseteq A_d$ and $D' \subseteq D_d$. It suffices to show that, with probability at least $1/6$, at least $a_d/4$ of the vertices in A_d are *not* in $H_v^{\ell+1}$. Assuming this outcome occurs, we have $\eta' = |A' \cup D'| \leq |A_d \cup D_d| - a_d/4 = r_d - a_d/4 \leq (7/8)r_d \leq (7/8)\eta$ as desired.

Let x be a random variable denoting the vertex in $A_d \cup D_d \cup \{v\}$ with highest priority. Consider the random process as follows: first toss a weighted coin to determine if x is in $A_d \cup \{v\}$ or $D_d \setminus A_d$, then select a vertex uniformly at random from the appropriate set. Since $|A_d| \geq |D_d|$, the former occurs with probability at least $1/2$. The remainder of the proof thus conditions on the assumption that $x \in A_d \cup \{v\}$, with the final success probability multiplied by $1/2$.

The remainder of the proof is similar in setup to Lemma 4.5, but the knocks-out relation and specific cases are more complicated. We say that x ***knocks out*** y if $x \not\preceq_d y$ or if both $x \preceq_d y$ and $y \preceq_d x$. Once more, it suffices to prove the following two claims: (1) If x knocks out y, then $y \notin H_v^{\ell+1}$, and (2) with probability at least $1/3$, x knocks out at least $a_d/4$ vertices from A_d.

Claim 1. We start by noting that by assumptions on choice of x, $f(v) = \rho(x)$. Moreover, since $\rho(x)$ is the highest priority v observes in either direction, $b(v) \leq \rho(x)$. A necessary condition for $y \in H_v^{\ell+1}$ is thus that $f(y) = \rho(x)$ and $b(y) \leq \rho(x)$. If $b(y) = f(y) = \rho(x)$, however, then y is marked for contraction and not part of $H_v^{\ell+1}$. So $y \in H_v^{\ell+1}$ also requires $b(y) < \rho(x)$.

By definition, if x knocks out y, then either $x \not\preceq_d y$, or both $x \preceq_d y$ and $y \preceq_d x$. If $x \not\preceq_d y$, then $f(y) \neq \rho(x)$, implying that $y \notin H_v^{\ell+1}$ as discussed above. Suppose instead that $x \preceq_d y$ and $y \preceq_d x$. Then $f(y) \geq \rho(x)$ and $b(y) \geq \rho(x)$. Again, this implies that $y \notin H_v^{\ell+1}$ because $b(y) \not< \rho(x)$.

Claim 2. If $x \not\preceq_d y$, then x knocks out y; likewise if $y \not\preceq_d x$, then y knocks out x. If $x \preceq_d y$ and $y \preceq_d x$, then x and y knock out each other. Therefore, either x knocks out y, y knocks out x, or both. Moreover, $x = v$ knocks out every vertex as in this case $f(v) = b(v) = \rho(v)$, and hence $H_v^{\ell+1}$ is the empty graph. The total number of pairs $x \in A_d \cup v$ and $(y \neq x) \in A_d$ for which x knocks out y is at least $\binom{a_d}{2} + a_d = a_d(a_d + 1)/2$. The rest of the claim is the same as Lemma 4.5, which completes the proof. □

The preceding lemma indicates that each level of recursion is likely to reduce the total number of active vertices by a constant factor. The following lemma applies this lemma across λ levels of recursion to conclude that v is likely to be in its own subproblem, or marked for contraction, before the recursion bottoms out.

Lemma 5.9. *Consider any vertex v and a complete execution of the recursive algorithm. With probability at least $1–32 \lg |V| /\lambda$, v has no active predecessors at the λ-th level of recursion.*

Proof. Let X_j be a random variable denoting the number of levels ℓ for which the number of active predecessors and active successors of v falls in the range $[(7/8)^{j+1}, (7/8)^j) \cdot |V|$, for integer j. When $j \geq \log_{8/7} |V|$, the high end of the range is strictly less than $1/|V| \cdot |V| = 1$, meaning that v has no active predecessors or active successors remaining. Let $X = \sum_{j<\log_{8/7}|V|} X_j$. If $X < \lambda$, then at or before the the λ-th level of recursion, v has no active predecessors. Our goal is thus to argue that this event is likely to occur. Lemma 5.8 says that the number of rounds necessary to get a $(7/8)$ reduction is at most six in expectation. Thus, $E[X_j] \leq E[X_j | X_j \geq 1] \leq 6$. By linearity of expectation, $E[X] \leq 6 \log_{8/7}(|V|) < 32 \lg(|V|)$. By Markov's inequality, $\mathbf{Pr}[X \geq \lambda] \leq 32 \lg(|V|)/(\lambda)$. □

5.4.3 Edges are Likely to Become Satisfied. We have argued that in each execution of the recursive algorithm, a particular edge (u, v) is unlikely to become broken, and moreover that v is likely to either be in its own subproblem or marked for contraction. The implication is that if both favorable outcomes occur, the edge (u, v) either crosses subproblem boundaries and becomes satisfied, or both u and v are contracted with each other and the edge disappears. Completing this argument again requires monotonic progress on satisfied edges. The following lemma says that satisfied edges never become violated later.

Lemma 5.10. *Consider an execution of the recursive algorithm RecurSCC. If an edge (u, v) is satisfied at the ℓ-th level of recursion, then it remains satisfied at all subsequent levels of recursion.*

Proof. Proof by induction on the level of recursion. Consider the call at the ℓ-th level of recursion, and suppose that the edge (u, v) is satisfied at the start of the call. The goal is to show that it remains satisfied in the next recursive subproblem. Note that once u and v fall in different subproblems (or the same subproblem marked for contraction), their relative order never changes.

We first claim that if $l(v) \geq l(u)$ and u and v are in the same subproblem, then (u, v) remains satisfied at the next level of recursion. To see that, if $l(v) > l(u)$, then sorting by label keeps the edge satisfied. If $l(v) = l(u)$ but (u, v) is satisfied initially, then $index(v) > index(u)$. Thus, breaking ties by index keeps the edge satisfied. The remainder of the proof thus focuses on showing that $l(v) \geq l(u)$.

Since (u, v) is satisfied, $u \preceq_0 v$. More importantly, for all x, $x \preceq_d u$ implies $x \preceq_d v$. Similarly, $v \preceq_d x$ implies $u \preceq_d x$. It follows that

$$f(u) \leq f(v) \text{ and } b(u) \geq b(v). \tag{1}$$

To show $l(v) \geq l(u)$, we consider several cases.

Case 1: $f(u) > b(u)$. Then $l(u) = f(u)$. We have $f(v) \geq f(u) > b(u) \geq b(v)$, where the first and last inequality follow Equation (1) and the middle one is by assumption. Both vertices are assigned their forwards label, and $f(v) \geq f(u)$, implying $l(v) \geq l(u)$.

Case 2: $f(u) = b(u)$. Then u is assigned label $l(u) = -b(u) + 1/2$. Similarly to the first case, we have $f(v) \geq f(u) = b(u) \geq b(v)$, implying $f(v) \geq b(v)$. If $f(v) > b(v)$, then $l(v) = f(v)$, which is nonnegative and hence larger than $l(u) = -b(u) + 1/2$. If $f(v) = b(v)$, then $l(v) = -b(v) + 1/2$. Since $b(u) \geq b(v)$, $l(v) = -b(v) + 1/2 \geq -b(u) + 1/2 = l(u)$.

Case 3: $f(u) < b(u)$. Then $l(u) = -b(u)$. If $f(v) > b(v)$, then $l(v) = f(v) > -b(u) = l(u)$, so $l(v) > l(u)$. If $f(v) = b(v)$, then $l(v) = -b(v) + 1/2$. Since $b(u) \geq b(v)$ from Equation (1), $-b(v) \geq -b(u)$, implying $l(v) = -b(v) + 1/2 \geq -b(u) = l(u)$. If $f(v) < b(v)$, then $l(v) = -b(v)$. Since $b(u) \geq b(v)$, $l(v) \geq l(u)$. □

Lemma 5.11. *Let (u, v) be any edge, and consider a complete execution of* RecurSCC *followed by the contraction step with parameters $\lambda \geq 128 \lg V$ and $K \geq 8\lambda \lg V$. If (u, v) is violated initially, then with probability at least $1/2$, (u, v) is not violated at the end (either because it is satisfied or removed from the graph). If (u, v) is not violated initially, then with probability 1 it is also not violated at the end.*

Proof. By Lemma 5.10, a satisfied edge always remains satisfied. It may, however, be removed from the graph in the contraction step. Nevertheless, it can never become violated. The remainder instead focuses on the case that an edge (u, v) is initially violated.

Let A be the event that (u, v) is violated at the end of the execution. Let B be the event that the edge breaks at some level of recursion, and let C be the event that the two endpoints u and v are in the same level-λ subproblem. If neither B nor C occurs, then either the edge crosses properly ordered subproblems at some level, or u and v are marked for contraction. In either case, (u, v) is not violated anymore. We thus have $\mathbf{Pr}[A] \leq \mathbf{Pr}[B] + \mathbf{Pr}[C]$ by a union bound.

By Lemma 5.9, $\mathbf{Pr}[C] \leq 32 \lg V/\lambda \leq 1/4$ for the specified choice of λ. By Lemma 5.5, the probability of breaking at any individual level is at most $2 \lg V/K$. Taking a union bound across λ levels, we have $\mathbf{Pr}[B] \leq 2\lambda \lg V/K \leq 1/4$ for the specified choice of K. Adding these together gives total failure probability of at most $1/2$. □

5.4.4 Bounds on the Main Algorithm.

Lemma 5.12. *Only vertices that are strongly connected are contracted.*

Proof. Given vertices u and v that are contracted, we will show that u and v are strongly connected. Since u and v are contracted, it must be the case that $f(u) = b(u) = f(v) = b(v)$. There must be some vertex x such that $\rho(x) = f(u)$. It could be the case that either u or v is vertex x. Since $\rho(x) = f(u) = b(u) = f(v) = b(v)$, $x \preceq_d u$, $x \preceq_d v$, $u \preceq_d x$, and $v \preceq_d x$. Therefore, x is strongly connected to both u and v, which implies that u and v are strongly connected. □

We next prove Theorem 5.1, which states that the algorithm topologically sorts the condensation of the graph after $\lceil (c + 2) \lg V \rceil$ executions with failure probability at most $1/V^c$, for any $c > 0$.

Proof of Theorem 5.1. By Lemma 5.12, the algorithm never performs any erroneous contractions. If the algorithm terminates, it must therefore be the case that the graph is topologically sorted, which is only possible if there are no cycles, i.e., if all strongly connected components have been contracted.

Lemma 5.10 says that violated edges are never introduced. Moreover, by Lemma 5.11, each violated edge has a constant probability of being removed or becoming satisfied. The rest of the proof is the same as Theorem 4.9.

6 CONCLUSIONS

This article has presented the first algorithm for topological sort and related problems that is I/O efficient even for sparse graphs.

The main question remaining is whether the algorithm can be improved to achieve an I/O cost $O(sort(E) \cdot \log^x V)$, for $x < 5$. One of the logarithmic factors arises from the fact that the number of distances K is large (i.e., $K = \Theta(\log^2 V)$). Another arises from the fact that distance ranges do not overlap at each level of recursion. We suspect that at least one of these logarithmic factors can be removed, yielding $x = 4$ or potentially even $x = 3$. Achieving $x < 3$, however, seems difficult. Inherent in the approach are at least two logarithmic factors: the number of iterations and the number of levels of recursion. Moreover, reducing the number of distances to $K = O(1)$, which would be necessary to get $x < 3$, would require some significant new ideas.

Another interesting question is whether randomization is necessary for these problems. Randomization plays a key role in our algorithm, but there may be alternative approaches.

REFERENCES

[1] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. 1998. A functional approach to external graph algorithms. In *Proceedings of the 6th Annual European Symposium on Algorithms*. 332–343. http://dl.acm.org/citation.cfm?id=647908.740141.

[2] Alok Aggarwal and Jeffrey Vitter. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM* 31, 9 (1988), 1116–1127.

[3] Deepak Ajwani, Adan Cosgaya-Lozano, and Norbert Zeh. 2011. Engineering a topological sorting algorithm for massive graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. 139–150.

[4] Lars Arge. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*. 334–345.

[5] Lars Arge, Gerth Stølting Brodal, and Laura Toma. 2004. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms* 53, 2 (2004), 186–206. https://doi.org/10.1016/j.jalgor.2004.04.001

[6] Lars Arge, Ulrich Meyer, and Laura Toma. 2004. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Prooceedings of the 31st International Colloquium on Automata, Languages and Programming*. 146–157.

[7] Lars Arge and Morten Revsbæk. 2009. I/O-efficient contour tree simplification. In *Proceedings of the International Symposium on Algorithms and Computation*. 1155–1165.

[8] Lars Arge, Laura Toma, and Norbert Zeh. 2003. I/O-efficient topological sorting of planar DAGs. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*. 85–93.

[9] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. 2000. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. 859–860. http://dl.acm.org/citation.cfm?id=338219.338650.

[10] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1995. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*. 139–149. http://dl.acm.org/citation.cfm?id=313651.313681.

[11] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2005. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. 735–744. http://dl.acm.org/citation.cfm?id=1070432.1070536

[12] Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. 2013. A labeling approach to incremental cycle detection. *CoRR* abs/1310.8381. arXiv:1310.8381. http://arxiv.org/abs/1310.8381.

[13] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. 2005. *A Divide-and-Conquer Algorithm for Identifying Strongly Connected Components*. Technical Report RC23744. IBM Research.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. Introduction to Algorithms (2nd ed.). MIT Press, Cambridge, MA, USA.

[15] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *Journal of the ACM* 44, 5 (1997), 669–696. https://doi.org/10.1145/265910.265914

[16] Jeremy T. Fineman. 2018. Nearly work-efficient parallel algorithm for digraph reachability. In *Proceedings of the 50th Annual ACM SIGACT Symposium on the Theory of Computation*. 457–470.

[17] Jelle Hellings, George H. L. Fletcher, and Herman Haverkort. 2012. Efficient external-memory bisimulation on DAGs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 553–564.

[18] Vijay Kumar and Eric J. Schwabe. 1996. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*. 169–176. http://dl.acm.org/citation.cfm?id=829517.830723.

[19] Anil Maheshwari and Norbert Zeh. 2008. I/O-efficient planar separators. *SIAM Journal on Computing* 38, 3 (2008), 767–801.

[20] Kurt Mehlhorn and Ulrich Meyer. 2002. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*. 723–735. http://dl.acm.org/citation.cfm?id=647912.740673.

[21] Ulrich Meyer and Norbert Zeh. 2003. I/O-efficient undirected shortest paths. In *Proceedings of the 11th Annual European Symposium on Algorithms*. 434–445.

[22] Ulrich Meyer and Norbert Zeh. 2006. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the 14th Annual European Symposium on Algorithms*. 540–551.

[23] Kameshwar Munagala and Abhiram Ranade. 1999. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*. 687–694. http://dl.acm.org/citation.cfm?id=314500.314891.

[24] Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.

[25] Jeffrey D. Ullman and Mihalis Yannakakis. 1991. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence* 3, 2 (1991), 331–360. https://doi.org/10.1007/BF01530929

[26] Jeffrey Scott Vitter. 2008. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science* 2, 4 (2008), 305–474.

[27] Norbert Zeh. [n. d.]. I/O-Efficient Graph Algorithms. Retrieved April 6, 2018 from http://cs.au.dk/~large/ioS05/Znotes.pdf.

[28] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, Lijun Chang, and Xuemin Lin. 2015. I/O efficient: Computing SCCs in massive graphs. *The VLDB Journal* 24, 2 (2015), 245–270. https://doi.org/10.1007/s00778-014-0372-z

Received March 2019; revised April 2020; accepted August 2020