Fully Dynamic Algorithms for Transitive Reduction

Gramoz Goranci
Faculty of Computer Science, University of Vienna, Austria

Adam Karczmarz
University of Warsaw, Poland

Ali Momeni
Faculty of Computer Science, UniVie Doctoral School Computer Science DoCS, University of Vienna, Austria

Nikos Parotsidis
Google Research, Zürich, Switzerland

Abstract

Given a directed graph G, a transitive reduction G^t of G (first studied by Aho, Garey, Ullman [SICOMP '72]) is a minimal subgraph of G that preserves the reachability relation between every two vertices in G.

In this paper, we study the computational complexity of transitive reduction in the dynamic setting. We obtain the first fully dynamic algorithms for maintaining a transitive reduction of a general directed graph undergoing updates such as edge insertions or deletions. Our first algorithm achieves $O(m + n \log n)$ amortized update time, which is near-optimal for sparse directed graphs, and can even support extended update operations such as inserting a set of edges all incident to the same vertex, or deleting an arbitrary set of edges. Our second algorithm relies on fast matrix multiplication and achieves $O(m + n^{1.585})$ *worst-case* update time.

2012 ACM Subject Classification Theory of computation → Sparsification and spanners

Keywords and phrases Spectral sparsification, Dynamic algorithms, (Directed) hypergraphs, Data structures

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.92

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2504.18161`

Funding *Adam Karczmarz*: Supported by the National Science Centre (NCN) grant number 2022/47/D/ST6/02184. Work partially done at IDEAS NCBR, Poland.

1 Introduction

Graph sparsification is a technique that reduces the size of a graph by replacing it with a smaller graph while preserving a property of interest. The resulting graph, often called a *sparsifier*, ensures that the property of interest holds if and only if it holds for the original graph. Sparsifiers have numerous applications, such as reducing storage needs, saving bandwidth, and speeding up algorithms by using them as a preprocessing step. Sparsification has been extensively studied for various basic problems in both undirected and directed graphs [4, 5, 8, 16, 50, 49]. In this paper, we focus on maintaining a sparsifier for the notion of transitive closure in dynamic directed graphs.

Computing the transitive closure of a directed graph (digraph) is one of the most basic problems in algorithmic graph theory. Given a graph $G = (V, E)$ with n vertices and m edges, the problem asks to compute for every pair of vertices s, t on whether t is reachable from s in G. The efficient computation of the transitive closure of a digraph has received much attention over the past decades. In dense graphs, due to the problem being equivalent

52nd International Colloquium on Automata, Languages, and Programming (ICALP 2025).
Editors: Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis
Article No. 92; pp. 92:1–92:20


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to Boolean Matrix Multiplication, the best known efficient algorithm runs in $O(n^\omega)$ time, where $\omega < 2.371552$ [51, 19, 54, 55]. In sparse graphs, transitive closure can be trivially computed in $O(nm)$ time[1].

In their seminal work, Aho, Garey, and Ullman [4] introduced the notion of transitive reduction; a *transitive reduction* of a digraph G is a digraph G^t on V with fewest possible edges that preserves the reachability information between every two vertices in G. Transitive reduction can be thought of as a sparsifier for transitive closure.

While the transitive reduction is known to be uniquely defined for a directed acyclic graph (DAGs), it may not be unique for general graphs due to the existence of strongly connected components (SCCs). For each SCC S there may exist multiple smallest graphs on S that preserve reachability among its vertices. One example of such a graph is a directed cycle on the vertices of S. Significantly, [4] showed that computing the transitive reduction of a directed graph requires asymptotically the same time as computing its transitive closure.

It is important to note that a transitive reduction with an asymptotically smaller size than the graph itself is not guaranteed to exist even if we allow introducing auxiliary vertices: indeed, any bipartite digraph G with n vertices on both sides equals its transitive closure and one needs at least n^2 bits to uniquely encode such a digraph. This is in contrast to e.g., equivalence relations such as strong connectivity where sparsification all the way down to linear size is possible.

In a DAG G, the same unique transitive reduction G^t could be equivalently defined as the (inclusion-wise) minimal subgraph of G preserving the reachability relation [4]. In some applications, having a sparsifier that remains a subgraph of the original graph might be desirable. Unfortunately, in the presence of cycles, if we insist on G^t being a subgraph of a G, then computing such a subgraph G^t of minimum possible size is NP-hard[2]. However, if we redefine G^t to be simply an *inclusion-wise minimal subgraph* of G preserving its reachability, computing it becomes tractable again[3], as a minimal strongly connected subgraph of a strongly connected graph can be computed in near-linear time [25, 27]. Throughout this paper, for our convenience, we will adopt this minimal subgraph-based definition of a transitive reduction G^t of a general digraph. At the same time, we stress that all our algorithms can also be applied to the original "minimum-size" definition [4] of G^t after an easy adaptation of the way reachability inside SCCs is handled.

The transitive reduction of a digraph finds applications across a multitude of domains such as the reconstruction of biological networks [13, 26] and other types of networks (e.g., [40, 34, 3]), in code analysis and debugging [57, 38], network analysis and visualization for social networks [21, 17], signature verification [29], serving reachability queries in database systems [31], large-scale parallel machine rescheduling problems [36], and many more. In some of these applications (e.g., [13, 26]), identifying and eliminating edges *redundant* from the point of view of reachability[4] is more critical than reducing the size of the graph and, consequently, the space required to store it.

In certain applications, one might need to compute the transitive reduction of dynamically evolving graphs, where nodes and edges are being inserted and deleted from the graph and the objective is to efficiently update the transitive reduction after every update. The naive way to do that is to recompute it from scratch after every update, which has total update

[1] Interestingly, shaving a logarithmic factor is possible here [15].
[2] G has a Hamiltonian cycle iff G^t is a cycle consisting of all vertices of G
[3] In fact, an inclusion-wise minimal transitive reduction can have at most n more edges than the minimum-size transitive reduction, hence the former is a 2-approximation of the latter in terms of size.
[4] i.e. that can be removed from the graph without affecting the transitive closure.

time $O(m \cdot \min(n^\omega, nm))$, or in other words, the algorithm has $O(\min(n^\omega, nm))$ amortized update time. This is computationally very expensive, though. It is interesting to ask whether a more efficient approach is possible.

The concept of dynamically maintaining the sparsifier of a graph is not new. Sparsifiers for many graph properties have been studied in the dynamic setting, where the objective is to dynamically maintain a sparse certificate as edges or vertices are being inserted and/or deleted to/from the underlying dynamic graph. To the best of our knowledge, apart from transitive reduction, other studies have mainly focused on sparsifiers for dynamic *undirected* graphs.

In this paper, we study fully dynamic sparsifiers for reachability; that is, for one of the most basic notions in directed graphs. In particular, we continue the line of work initiated by La Poutré and van Leeuwen [41] who were the first to study the maintenance of the transitive reduction in the partially dynamic setting, over 30 years ago. They presented an incremental algorithm with total update time $O(mn)$, and a decremental algorithm with total update time that is $O(m^2)$ for general graphs, and $O(mn)$ for DAGs.

1.1 Our results

We introduce the first fully dynamic data structures designed for maintaining the transitive reduction in digraphs. These data structures are tailored for both DAGs and general digraphs, and are categorized based on whether they offer worst-case or amortized guarantees on the update time.

Amortized times for handling updates

Our first contribution is two fully dynamic data structures for maintaining the transitive reduction of DAGs and general digraphs, each achieving roughly linear amortized update time on the number of edges. Both data structures are combinatorial and deterministic, with their exact guarantees summarized in the theorems below.

▶ **Theorem 1.1.** *Let G be an initially empty graph that is guaranteed to remain acyclic throughout any sequence of edge insertions and deletions. Then, there is a fully dynamic deterministic data structure that maintains the transitive reduction G^t of G undergoing a sequence of edge insertions centered around a vertex or arbitrary edge deletions in $O(m)$ amortized update time, where m is the current number of edges in the graph.*

For general sparse digraphs, we obtain a much more involved data structure, where we pay an additional logarithmic factor in the update time.

▶ **Theorem 1.2.** *Given an initially empty general digraph G, there is a fully dynamic deterministic data structure that supports edge insertions centered around a vertex and arbitrary edge deletions, and maintains a transitive reduction G^t of G in $O(m + n \log n)$ amortized update time, where m is the current number of edges in the graph.*

In fact, the data structures from Theorems 1.1 and 1.2 support more general update operations: 1) insertions of a set of any number of edges, all incident to the same single vertex, and 2) the deletion of an arbitrary set of edges from the graph. Note that these *extended* update operations are more powerful compared to the single edge insertions and deletions supported by more traditional dynamic data structures. For further details, we refer the reader to Section 5 and the full version of the paper.

For sparse digraphs, our dynamic algorithms supporting insertions in $O(m + n \log n) = O(n \log n)$ are almost optimal, up to a $\log n$ factor. This is because a polynomially faster dynamic algorithm would lead to an improvement in the running time of the best-known static $O(n^2)$ algorithm for computing the transitive reduction of a sparse graph, which would constitute a major breakthrough.

Observe that within $O(m)$ amortized time spent on updating the data structure, one can explicitly list each edge of the maintained transitive reduction which is guaranteed to have at most m edges. This is why Theorems 1.1 and 1.2 do not come with separate query bounds.

Worst-case times for handling updates

Our second contribution is another pair of fully dynamic data structures that maintain the transitive reduction of DAGs and general digraphs. These data structures achieve *worst-case* update time (on the number of nodes) for vertex updates and sub-quadratic *worst-case* update time for edge updates. This is as opposed to Theorems 1.1 and 1.2, where the worst-case cost of a single update can be as much as $O(nm) = O(n^3)$.

Both data structures rely on fast matrix multiplication and are randomized, with their exact guarantee summarized in the theorem below.

▶ **Theorem 1.3.** *Let G be a graph that is guaranteed to remain acyclic throughout any sequence of updates. Then, there are randomized Monte Carlo data structures for maintaining the transitive reduction G^t of G*

- *in $O(n^2)$ worst-case update time for vertex updates, and*
- *in $O(n^{1.528} + m)$ worst-case update time for edge updates.*

Both data structures output a correct transitive reduction with high probability. They can be initialized in $O(n^\omega)$ time.

For general digraphs, the runtime guarantees for vertex updates remain the same, whereas for edge updates, we incur a slightly slower sub-quadratic update time.

▶ **Theorem 1.4.** *Given a general digraph G, there are randomized Monte Carlo data structures for maintaining the transitive reduction G^t of G*

- *in $O(n^2)$ worst-case update time for vertex updates, and*
- *in $O(n^{1.585} + m)$ worst-case update time for edge updates.*

Both data structures output a correct transitive reduction with high probability. They can be initialized in $O(n^\omega)$ time.

All our data structures require $O(n^2)$ space, similarly to the partially dynamic data structures proposed by [41]. Going beyond the quadratic barrier in space complexity is known to be a very challenging task in data structures for all-pairs reachability. For example, to the best of our knowledge, it is not even known whether there exists a *static* data structure with $O(n^{2-\epsilon})$ space and answering arbitrary-pair reachability queries in $O(m^{1-\epsilon})$ time. Finally, recall that in certain applications eliminating redundant edges in a time-efficient manner is vital, and quadratic space is not a bottleneck.

1.2 Related work

Due to their wide set of applications, sparsification techniques have been studied for many problems in both undirected and directed graphs. For undirected graphs, some notable examples include sparsification for k-connectivity [27, 37], shortest paths [5, 39, 7], cut sparsifiers [8, 9, 23], spectral sparsifiers [49, 50], and many more. For directed graphs,

applications of sparsification have been studied for reachability [4], strong connectivity [27, 56], shortest paths [32, 43], k-connectivity [24, 35, 16], cut sparsifiers [14], spectral sparsifiers for Eulerian digraphs [18, 46], and many more.

There is also a large body of literature on maintaining graph sparsifiers on dynamic undirected graphs. Examples of this body of work includes dynamic sparsifiers for connectivity [28], shortest paths [6, 22, 10], cut and spectral sparsifiers [2, 12], and k-edge-connectivity [1].

1.3 Organization

In Section 2, we set up some notation. Section 3 provides a technical overview of our algorithms for both DAGs and general graphs. We then present the simpler data structures for DAGs in Sections 4 and 5. Due to space constraints, the detailed description of our data structures for general graphs, together with some proofs, is deferred to the appendix.

2 Preliminaries

In this section, we introduce some notation and review key results on transitive reduction in directed graphs, which will be useful throughout the paper.

Graph Theory

Let $G = (V, E)$ be a directed, unweighted, and loop-less graph where $|V| = n$ and $|E| = m$. For each edge xy, we call y an *out-neighbor* of x, and x an *in-neighbor* of y. A *path* is defined as a sequence of vertices $P = \langle v_1, v_2, \ldots, v_k \rangle$ where $v_i v_{i+1} \in E$ for each $i \in [k-1]$. We call v_1 and v_k as the *first* vertex and the *last* vertex of P, respectively. The *length* of a path P, $|P|$, is the number of its edges. For two (possibly empty) paths $P = \langle u_1, u_2, \ldots, u_k \rangle$ and $Q = \langle v_1, v_2, \ldots, v_l \rangle$, where $u_k = v_1$, the concatenation of P and Q is the path obtained by identifying the last vertex of P with the first vertex of Q. i.e. the path $\langle u_1, u_2, \ldots, u_k = v_1, v_2, \ldots, v_l \rangle$. A *cycle* is a path whose first and last vertices are the same, i.e., $v_1 = v_k$. We say that G is a *directed acyclic graph* (DAG) if G does not have any cycle. We say there is a path $u \to v$ from u to v (or u can reach v) if there exists a path $P = \langle v_1, v_2, \ldots, v_k \rangle$ with $v_1 = u$ and $v_k = v$.

A graph H is called a *subgraph* of G if H can be obtained from G by deleting (possibly empty) subsets of vertices and edges of G. For a set $U \subseteq E$, we define $G \setminus U$ as the subgraph of G obtained by deleting edges in U from G. We also define $G \setminus uv = G \setminus \{uv\}$.

Transitive Reduction

A transitive reduction of a graph $G = (V, E)$ is a graph $G^t = (V, \tilde{E})$ with the fewest possible edges that preserves the reachability information between every two vertices in G. i.e., for arbitrary vertices $u, v \in V$, there is a $u \to v$ path in G iff there is a $u \to v$ path in G^t. Note that G^t may not be unique and might not necessarily be a subgraph of G.

For a DAG G, G has a unique transitive reduction G^t [4]. They presented an algorithm to compute G^t by identifying and eliminating the *redundant* edges. We say that edge $xy \in E$ is redundant if there is a directed path $x \to y$ in $G \setminus xy$.

▶ **Theorem 2.1** ([4]). *Every* DAG G *has a unique transitive reduction* G^t *that is a subgraph of* G *and is computed by eliminating all redundant edges of* G.

For a general graph G, a transitive reduction G^t can be obtained by replacing every strongly connected component (SCC) in G with a cycle and removing the redundant inter-SCC edges one-by-one [4]. But if we insist on G^t being a subgraph of G, then finding G^t is NP-hard since G has a Hamiltonian cycle iff G^t is a cycle consisting of all vertices of G. To overcome this theoretical obstacle, we consider a *minimal* transitive reduction G^t of G. Given a graph G, we call G^t a minimal transitive reduction of G if removing any edge from the subgraph G^t results in G^t no longer being a transitive reduction of G. For the rest of this paper, we assume G^t is a subgraph of G. At the same time, we once again stress this is merely for convenience and all our algorithms can be easily adapted to maintain the minimum-size reachability preserver with SCCs replaced with cycles.

3 Technical overview

Dynamic transitive closure has been extensively studied, with efficient combinatorial [42, 45] and algebraic [53, 47] data structures known. As mentioned before, computing transitive reduction is closely related to computing the transitive closure [4]. This is why, in this work, we adopt the general approach of reusing some of the technical ideas developed in the prior literature on dynamic transitive closure. The main challenge lies in our goal to achieve near-linear update time in the number of edges m and *constant* query time, or explicit maintenance of the transitive reduction. Existing dynamic transitive closure data structures such as those in [42, 47] have either $O(n^2)$ update time and $O(1)$ arbitrary-pair query time (which is optimal if the reachability matrix is to be maintained explicitly), or polynomial query time [53, 45, 47].

To maintain the transitive reduction, we do not need to support arbitrary reachability queries. Instead, we focus on maintaining specific reachability information between m pairs of vertices connected by an edge. This reachability information, however, is more sophisticated than in the case of transitive closure.

3.1 DAGs

Let us first consider the simpler case when G is a DAG. Recall (Theorem 2.1) that the problem boils down to identifying redundant edges. To test the redundancy of an edge uv, we need to maintain information on whether a $u \to v$ path exists in the graph $G \setminus uv$.

3.1.1 A reduction to the transitive closure problem

In acyclic graphs, a reduction resembling that of [4] (see Lemma 4.1) carries quite well to the fully dynamic setting. Roughly speaking, one can set up a graph G' with two additional copies of every vertex and edge, so that for all $u, v \in V$, paths $u'' \to v''$ between the respective second-copy vertices u'', v'' in G' correspond precisely to *indirect* paths $u \to v$ in G, i.e., $u \to v$ paths avoiding the edge uv. Clearly, an edge xy is redundant if indirect $x \to y$ paths exist. As a result, one can obtain a dynamic transitive reduction data structure by setting up a dynamic transitive closure data structure on G and issuing m reachability queries after each update. The reduction immediately implies an $O(n^2)$ worst-case update bound (Monte Carlo randomized) and $O(n^2)$ amortized update bound (deterministic) for the problem in the acyclic case via [42, 47], even in the case of vertex updates. This is optimal for dense acyclic graphs, at least if one is interested in maintaining the reduction explicitly.[5]

[5] Indeed, consider a graph $G = (V_1 \cup V_2 \cup \{s, t\}, E)$ with $E = (V_1 \times V_2) \cup \{vs : v \in V_1\} \cup \{tv : v \in V_2\}$, where $n = |V_1| = |V_2|$. No edges in this acyclic graph are redundant. However, adding the edge st

The reduction works best if the transitive closure data structure can maintain some (mostly fixed) m reachability pairs $Y \subseteq V \times V$ of interest more efficiently than via issuing m reachability queries. In our use case, the reachability pairs Y correspond to the edges of the graph, so an update of G can only change Y by one pair in the case of single-edge updates, and by n pairs sharing a single vertex in the case of vertex updates to G. Some algebraic dynamic reachability data structures based on dynamic matrix inverse [47, 48] come with such a functionality out of the box. By applying these data structures on top of the reduction, one can obtain an $O(n^{1.528} + m)$ worst-case bound for updating the transitive reduction of an acyclic digraph (see Theorem 6.4 for details). Interestingly, the $n^{1.528}$ term (where the exponent depends on the current matrix multiplication exponent [55] and simplifies to $n^{1.5}$ if $\omega = 2$) typically emerges in connection with single-source reachability problems, and $O(n^{1.528})$ is indeed conjectured optimal for fully dynamic single-source reachability [53]. One could say that our algebraic transitive reduction data structure for DAGs meets a fairly natural barrier.

3.1.2 A combinatorial data structure with amortized linear update time bound

The algebraic reachability data structures provide worst-case guarantees but are more suitable for denser graphs. In particular, they never achieve near-linear in n update bounds, even when applied to sparse graphs. On the other hand, some combinatorial fully dynamic reachability data structures, such as [44], do achieve near-linear in n update bound. However, these update bound guarantees (1) are only amortized, and (2) come with a non-trivial polynomial query procedure that does not inherently support maintaining a set Y of reachability pairs of interest. This is why relying on the reduction (Lemma 4.1) does not immediately improve the update bound for sparse graphs. Instead, we design a data structure tailored specifically to maintain the transitive reduction of a DAG (Theorem 1.1). We later extend it to general graphs, ultimately obtaining Theorem 1.2.

First of all, we prove that given a source vertex s of a DAG G, one can efficiently maintain, subject to edge deletions, whether an indirect $s \to t$ path exists in G for every outgoing edge $st \in E$ (Lemma 5.1). This "indirect paths" data structure is based on extending the decremental single-source reachability data structure of [30], which is a classical combinatorial data structure with an optimal $O(m)$ total update time.

Equipped with the above, we apply the strategy used in the reachability data structures of [44, 45]: every time an insertion of edges centered at a vertex z issued, we build a new decremental single-source indirect paths data structure D_z "rooted" at z and initialized with the current snapshot of G. Such a data structure will not accept any further insertions, so the future insertions to G are effectively ignored by D_z. Intuitively, the responsibility of D_z is to handle (i.e., test the indirectness of) paths in the *current* graph G *whose most recently updated vertex is* z.[6] This way, handling each path in G is delegated to precisely one maintained decremental indirect paths data structure rooted at a single vertex.

Compared to the data structures of [44, 45], an additional layer of global bookkeeping is used to aggregate the counts of alternative paths from the individual data structures D_z. That is, for an arbitrary $z \in V$, D_z contributes to the counter iff it contains an alternative

makes all n^2 edges $V_1 \times V_2$ redundant. Adding and removing st back and forth causes $\Theta(n^2)$ amortized change in the transitive reduction.

[6] In the transitive closure data structures [44, 45], the concrete goal of an analogous data structure rooted at z is to efficiently query for the existence of a path $x \to y$, where z is the most recent updated vertex along the path.

path. Using this technique, if the count is zero for some edge uv, then uv is not redundant. This allows constant-time redundancy queries, or, more generally, explicitly maintaining the set of edges in the transitive reduction.

We prove that all actions we perform can be charged to the $O(m)$ initialization time of the decremental data structures D_z. Since, upon each insert update (incident to the same vertex), we (re)initialize only one data structure D_z, the amortized update time of the data structure is $O(m)$.

3.2 General graphs

For maintaining a transitive reduction of a general directed graph, the black-box reduction to fully dynamic transitive closure (Lemma 4.1) breaks. A natural idea to consider is to apply it to the acyclic *condensation* of G obtained by contracting the strongly connected components (SCCs). However, a single edge update to G might change the SCCs structure of G rather dramatically.[7] This, in turn, could enforce many single-edge updates to the condensation, not necessarily centered around a single vertex. Using a dynamic transitive closure data structure (accepting edge updates) for maintaining the condensation in a black-box manner would then be prohibitive: all the known fully dynamic transitive closure data structures have rather costly update procedures, which are at least linear in n. Consequently, handling the (evolving) SCCs in a specialized non-black-box manner seems inevitable here.

Nevertheless, using the condensation of G may still be useful. First, since we are aiming for near-linear (in m) update time anyway (recall that $O(m^{1-\epsilon})$ update time is likely impossible), the $O(n)$-edge transitive reductions of the individual SCCs can be recomputed from scratch. This is possible since a minimal strongly connected subgraph of a strongly connected graph can be computed in $O(m + n \log n)$ time [25].

The above allows us to focus on detecting the redundant *inter-SCC* edges xy, that is, edges connecting two different SCCs X, Y of G. The edge xy can be redundant for two reasons. First, if there exists an indirect $X \to Y$ path in the condensation of G, then xy is redundant by the arguments we have used for DAGs. Second, if there are other *parallel* edges $x_1y_1, \ldots, x_ky_k$ such that $x_i \in X$ and $y_i \in Y$. In the latter case, if there is no indirect path $X \to Y$, then clearly the transitive reduction contains precisely one of $xy, x_1y_1, \ldots, x_ky_k$. In other words, all these edges but one (not necessarily xy) are redundant.

3.2.1 Extending the combinatorial data structure

Extending the data structure for DAGs to support SCCs involves addressing some technical challenges. The decremental indirect path data structures need to be generalized to support detecting indirect paths in the condensation. The approach of [30] breaks for general graphs, though. One solution here would be to adapt the near-linear decremental single-source reachability for general graphs [11]. However, that data structure is randomized, slower by a few log factors, and much more complex. Instead, as in [44, 45], we take advantage of the fact that we always maintain n decremental indirect paths data structures on a nested family of snapshots of G. This allows us to apply an algorithm from [45] to compute how the SCCs decompose as a result of deleting some edges in *all* the snapshots at once, in $O(m + n \log n)$ time. Since the SCCs in snapshots decompose due to deletions, the condensations are not completely decremental in the usual sense: some intra-SCC edges may

[7] For example, a single vertex of the condensation may break into n vertices forming a path (consider removing a single edge of an induced cycle).

turn into inter-SCC edges and thus are added to the condensation. Similarly, the groups of parallel inter-SCC edges may split, and some edges that have previously been parallel may lose this status. Nevertheless, we show that all these problems can be efficiently addressed within the amortized bound of $O(m + n \log n)$ which matches that of the fully dynamic reachability data structure of [45]. Similarly as in DAGs, one needs to check $O(1)$ counters and flags to test whether some edge of G is redundant or not; this takes $O(1)$ time. The details can be found in the full version of the paper.

3.2.2 Inter-SCC edges in algebraic data structures for general graphs

As indicated previously, operating on the condensation turns out to be very hard when using the dynamic matrix inverse machinery [53, 47]. Intuitively, this is because these data structures model edge updates as entry changes in the adjacency matrix, and this is all the dynamic matrix inverse can accept. It seems that if we wanted an algebraic data structure to maintain the condensation, we would need to update it explicitly by issuing edge updates. Recall that there could be $\Theta(n)$ such edge updates to the condensation per single-edge update issued to G, even if G is sparse. This is prohibitive, especially since any updates to a dynamic matrix inverse data structure take time superlinear in n.

To deal with these problems, we refrain from maintaining the condensation. Instead, we prove an algebraic characterization of the redundancy of a group of parallel inter-SCC edges $F = \{u_1v_1, \ldots, u_kv_k\}$ connecting two distinct SCCs R, T of G. Specifically, we prove that if $\tilde{A}(G)$ is the *symbolic adjacency matrix* [47] (i.e., a variant of the adjacency matrix with each 1 corresponding to an edge uv is replaced with an independent indeterminate $x_{u,v}$), then in order to test whether F is redundant it is enough to verify a certain polynomial identity (see the full version of the paper) on some $k + 1$ elements of the inverse $(\tilde{A}(G))^{-1}$ whose elements are degree $\leq n$ multivariate polynomials. Consequently, through all groups F of parallel inter-SCC edges in G, we obtain that $O(n + m)$ elements of the inverse have to be inspected to deduce which inter-SCC edges are redundant.

By a standard argument involving the Schwartz-Zippel lemma, in the implementation, we do not actually need to operate on polynomials (which may contain an exponential number of terms of degree $\leq n$). Instead, for high-probability correctness it is enough to test which of the aforementioned identities hold for some random variable substitution from a sufficiently large prime field $\mathbb{Z}/p\mathbb{Z}$ where $p = \Theta(\text{poly}\,(n))$. Since the inverse of a matrix can be maintained explicitly in $O(n^2)$ worst-case time subject to row or column updates, this yields a transitive reduction data structure with $O(n^2)$ worst-case bound per vertex update (see the full version of the paper).

Obtaining a better worst-case bound for sparser graphs in the single-edge update setting is more challenging. Unfortunately, the elements of the inverse used for testing the redundancy of a group F of parallel intra-SCC edges do not quite correspond to the individual edges of F; they also depend, to some extent, on the global structure of G. Specifically, the aforementioned identity associated with F involves elements $(r, u_1), \ldots, (r, u_k)$, $(v_1, t), \ldots, (v_k, t)$, and (r, t) of the inverse, where r, t are arbitrarily chosen *roots* of the SCCs R and T, respectively.

Dynamic inverse data structures with subquadratic update time [53, 47] (handling single-element matrix updates) generally do not allow accessing arbitrary query elements of the maintained inverse in constant time; this costs at least $\Theta(n^{0.5})$ time. They can, however, provide constant-time access to a specified *subset of interest* $Y \subseteq V \times V$ of its entries, at the additive cost $O(|Y|)$ in the update bound. Ideally, we would like Y to contain all the $O(m + n)$ elements involved in identities to be checked. However, the mapping of vertices V

to the roots of their respective SCCs may quickly become invalid due to adversarial edge updates causing SCC splits and merges. Adding new entries to Y is costly, though: e.g., inserting n new elements takes $\Omega(n^{1.5})$ time.

We nevertheless deal with the outlined problem by applying the hitting set argument [52] along with a heavy-light SCC distinction. For a parameter $\delta \in (0,1)$, we call an SCC heavy if it has $\Theta(n^\delta)$ vertices, and light otherwise. We make sure that at all times our set of interest Y in the dynamic inverse data structure contains

1. the edges E,
2. a sample $(S \times V) \cup (V \times S)$ of rows and columns for a random subset S (sampled once) of size $\Theta(n^{1-\delta} \log n)$, and
3. all $O(n^{1+\delta})$ pairs (u, v) such that u and v are both in the same *light* SCC.

This guarantees that the set Y allows for testing all the required identities in $O(n+m)$ time, has size $\tilde{O}(n^{1+\delta} + n^{2-\delta} + m)$ and evolves by $O(n^{1+\delta})$ elements per single-edge update, all aligned in $O(n)$ small square submatrices. For an optimized choice of δ, the worst-case update time of the data structure is $O(n^{1.585} + m)$, i.e., slightly worse than we have achieved for DAGs.

It is an interesting problem whether the transitive reduction of a general directed graph can be maintained within the natural $O(n^{1.528} + m)$ worst-case bound per update. The details can be found in the full version of the paper.

4 Reductions

In this section, we provide a reduction from fully dynamic transitive reduction to fully dynamic transitive closure on DAGs.

▶ **Lemma 4.1.** *Let $G = (V, E)$ be a fully dynamic digraph that is always acyclic. Let $Y = \{(x_i, y_i) : i = 1, \ldots, k\} \subseteq V \times V$. Suppose there exists a data structure maintaining whether paths $x_i \to y_i$ exist in G for $i = 1, \ldots, k$ subject to either:*

- *fully dynamic single-edge updates to G and single-element updates to Y,*
- *fully dynamic single-vertex updates to G (i.e. changing any number of edges incident to a single vertex v) and single-vertex updates to Y (i.e., for some $v \in V$, inserting/deleting from Y any number of pairs (x, y) satisfying $v \in \{x, y\}$).*

Let $T(n, m, k)$ be the (amortized) update time of the assumed data structure.

Then, there exists a fully dynamic transitive reduction data structure supporting the respective type of updates to G with $O(T(n, m, m))$ amortized update time.

Proof. Let $V' = \{v' : v \in V\}$ and $V'' = \{v'' : v \in V\}$ be two copies of the vertex set V. Let $E' = \{uv' : uv \in E\}$ and $E'' = \{u'v'' : uv \in E\}$. Consider the graph

$$G' = (V \cup V' \cup V'', E \cup E' \cup E'').$$

Note that a single-edge (resp., single-vertex) update to G can be translated to three updates of the respective type issued to G'.

Let us now show that an edge $xy \in E$ is redundant in G if and only if there is a path $x \to y''$ in G'. Since G is a DAG, if xy is redundant, there exists an $x \to y$ path in G with at least two edges, say $Q \cdot wz \cdot zy$, where $Q = x \to w$ (possibly $w = x$). By the construction of G', $wz', z'y'' \in E(G')$, and $Q \subseteq G \subseteq G'$. As a result, x can reach y'' in G'. Now suppose that there is a path from x to y'' in G'. By the construction of G', such a path is of the form $Q \cdot wz' \cdot z'y''$ (where $Q \subseteq G$) since V'' has incoming edges only from V', and $V' \cup V''$ has incoming edges only from V. The $x \to y$ path $Q \cdot wz \cdot zy \subseteq G$ has at least two edges and $x \neq z$ (by acyclicity), so xy is indeed a redundant edge.

It follows that the set of redundant edges of G can be maintained using a dynamic transitive closure data structure set up on the graph G' with the set Y (of size $k = m$) equal to $\{uv'' : uv \in E\}$. To handle a single-edge update to G, we need to issue three single-edge updates to the data structure maintaining G', and also issue a single-element update to the set Y. Analogously for vertex updates: an update centered at v causes an update to Y such that all inserted/removed elements have one of its coordinates equal to v or v''. ◀

5 Combinatorial Dynamic Transitive Reduction on DAGs

In this section, we explain a data structure for maintaining the transitive reduction of a DAG G, as summarized in Theorem 1.1 below. The data structure supports *extended* update operations, i.e., it allows the deletion of an *arbitrary* set of edges E_{del} or the insertion of some edges E_u incident to a vertex u, known as the center of the insertion.

The data structure for general graphs is explained in the full version of the paper.

▶ **Theorem 1.1.** *Let G be an initially empty graph that is guaranteed to remain acyclic throughout any sequence of edge insertions and deletions. Then, there is a fully dynamic deterministic data structure that maintains the transitive reduction G^t of G undergoing a sequence of edge insertions centered around a vertex or arbitrary edge deletions in $O(m)$ amortized update time, where m is the current number of edges in the graph.*

The data structure uses a straightforward idea to maintain G^t: an edge xy does not belong to G^t if y has an in-neighbor $z \neq x$ reachable from x. Thus, one can maintain G^t by maintaining the reachability information for each vertex $u \in V$, but naively maintaining them results in $O(mn)$ amortized update time for the data structure.

To improve the amortized update time to $O(m)$, we maintain the reachability information on a subgraph $G^u = (V, E^u)$ for every vertex $u \in V$ defined as the snapshot of G taken after the last insertion E_u centered around u (if such an insertion occurred). The reachability information are defined as follows: Desc^u is the set of vertices reachable from u in G^u and Anc^u is the set of vertices that can reach u in G^u.

Note that subsequent edge insertions centered around vertices *different* from u do not change G^u. However, edges that are subsequently deleted from G are also deleted from G^u, ensuring that at each step, $E^u \subseteq E$. Therefore, G^u undergoes only edge deletions while we decrementally maintain Desc^u and Anc^u until an insertion E_u centered around u happens and we reinitialize G^u.

To decrementally maintain Desc^u and Anc^u, our data structure uses an extended version of the decremental data structure of Italiano [30], summarized in the lemma below. See the full version of the paper for a detailed discussion.

▶ **Lemma 5.1.** *Given a DAG $G = (V, E)$ initialized with the set Desc^r of vertices reachable from a root vertex r and the set Anc^r of vertices that can reach r, there is a decremental data structure undergoing arbitrary edge deletions at each update that maintains Desc^r and Anc^r in $O(1)$ amortized update time.*

Additionally, the data structure maintains the sets D^r and A^r of vertices removed from Desc^r and Anc^r, resp., due to the last update and supports the following operations in $O(1)$ time:

- $\textsc{In}(y)$: *Return* `True` *if $y \neq r$ and y has an in-neighbor from $\text{Desc}^r \setminus r$, and* `False` *otherwise.*
- $\textsc{Out}(x)$: *Return* `True` *if $x \neq r$ and x has an out-neighbor to $\text{Anc}^r \setminus r$, and* `False` *otherwise.*

The lemma below shows how maintaining Desc^u and Anc^u will be useful in maintaining G^t. Recall that an edge $xy \in E$ is redundant if there exists a $x \to y$ path in $G \setminus xy$.

▶ **Lemma 5.2.** *Edge xy is redundant in G iff one of the following holds.*
1. *There is a vertex $z \notin \{x, y\}$ such that $x \in \text{Anc}^z$ and $y \in \text{Desc}^z$ in G^z.*
2. *Vertex y has an in-neighbor $z \in \text{Desc}^x \setminus x$ in G^x.*
3. *Vertex x has an out-neighbor $z \in \text{Anc}^y \setminus y$ in G^y.*

Proof. We first show the "if" direction. Suppose that xy is redundant in G. Since G is a DAG, and by the definition of redundant edges, there exists a directed path P from x to y in $G \setminus xy$ whose length is at least two. Let c be a vertex on P that has been a center of an insertion most recently. This insert operation constructed sets Desc^c and Anc^c. At the time of this insertion, all the edges of the path P were already present in the graph, and thus P exists in the graph G^c. Now, if $c \notin \{x, y\}$, then P is a concatenation of subpaths $x \to c$ and $c \to y$ in G^c, or equivalently, Item (1) holds. If $c = x$, then Item (2) holds, and if $c = y$, then Item (3) holds.

For the "only-if" direction, we only prove Item (2); items (1) and (3) can be shown similarly. To prove Item (2), let us assume that y has an in-neighbor $z \in \text{Desc}^x \setminus x$ in G^x. Our goal is to show that xy is redundant in G. To this end, since $z \notin \{x, y\}$ and G^x is a DAG, there exists a path P from x to y in G^x that is a concatenation of the subpath $x \to z$ and the edge zy. Note that P has length at least two and does not contain the edge xy. Since $E^x \subseteq E$, P is also a path from x to y in $G \setminus xy$, which in turn implies that xy is redundant in G. ◀

To incorporate Lemma 5.2 in our data structure, we define $\text{c}(xy)$ and $\text{t}(xy)$ for every edge $xy \in E$ as follows:

- The counter $\text{c}(xy)$ stores the number of vertices $z \notin \{x, y\}$ such that $xy \in E^z$, $x \in \text{Anc}^z$, and $y \in \text{Desc}^z$ in G^z.
- The binary value $\text{t}(xy)$ is set to 1 if either y has an in-neighbor $z \in \text{Desc}^x \setminus x$ in G^x or x has an out-neighbor $z \in \text{Anc}^y \setminus y$ in G^y, and is set to 0 otherwise.

Note that for a redundant edge xy in G, a vertex z contributes towards $\text{c}(xy)$ iff $xy \in E^z$. If no such z exists, then xy has become redundant due to the last insertion being centered around x or y, which in turn implies $\text{t}(xy) = 1$. Combining these two facts, we conclude the following invariant.

▶ **Invariant 5.1.** *An edge $xy \in E$ belongs to the transitive reduction G^t iff $c(xy) = 0$ and $t(xy) = 0$.*

In the rest of the section, we show how to efficiently maintain $\text{c}(xy)$ and $\text{t}(xy)$ for every $xy \in E$.

5.1 Edge insertions

After the insertion of E_u centered around u, the data structure updates G^u, while leaving every other graph G^v is *unchanged*.

We simply use a graph search algorithm to recompute Desc^u and Anc^u, and the sets C^u and B^u of the new vertices added to Desc^u and Anc^u, respectively, due to the insertion of E_u.

To maintain $\text{c}(xy)$ for an edge $xy \in E$, we only need to examine the contribution of u in $\text{c}(xy)$ as only G^u has changed. By definition, every edge xy touching u, i.e., with $x = u$ or $y = u$, does not contribute in $\text{c}(xy)$. Note that E^u consists of the edges before the update and the newly added edges, which leads to distinguishing the following cases.

1. Suppose that xy is not touching u and has already existed in E^u before the update. Then, $\mathrm{c}(xy)$ will increase by one only if the update makes x to reach y through u. i.e., there is no path $x \to u \to y$ before the update ($x \notin \mathrm{Anc}^u \setminus \mathrm{B}^u$ or $y \notin \mathrm{Desc}^u \setminus \mathrm{C}^u$), but there is at least one afterwards (i.e., $x \in \mathrm{Anc}^u$ and $y \in \mathrm{Desc}^u$).
2. Suppose that xy is not touching u and has added to E^u after the update. Since this is the first time we examine the contribution of u towards $\mathrm{c}(xy)$, we increment $\mathrm{c}(xy)$ by one if x can reach y through u (i.e., $x \in \mathrm{Anc}^u$ and $y \in \mathrm{Desc}^u$).

We now maintain $\mathrm{t}(xy)$ for an edge $xy \in E$. By definition, $\mathrm{t}(xy)$ could be affected only if xy touches u. Since G^u undergoes edge insertions, $\mathrm{t}(xy)$ can only change from 0 to 1. For every edge $xy \in E$ touching u, we set $\mathrm{t}(xy) \leftarrow 1$ if one of the following happen:

- $x = u$ and y has an in-neighbor $z \neq u$ in G^u reachable from u (i.e., if $\textsc{In}(y)$ reports `True`), or
- $y = u$ and x has an out-neighbor $z \neq u$ in G^u that can reach y (i.e., if $\textsc{Out}(y)$ reports `True`).

Note that both cases above try to insure the existence of a path $x \to z \to y$ when $x = u$ or $y = u$.

▶ **Lemma 5.3.** *After each insertion, the data structure maintains the transitive reduction G^t of G in $O(m)$ worst-case time, where m is the number of edges in the current graph G.*

Proof. Suppose that the insertion of edges E_u has happened centered around u.

Correctness. By construction, all possible scenarios for the edge xy are covered and the values $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ are correctly maintained after each insert update with respect to G^u for every edge $xy \in E$. Since G^u is the only graph changing during the update, it follows that $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ are correctly maintained with respect to G^v, $v \in V$. The correctness immediately follows from Invariant 5.1 and Lemma 5.2.

Update time. The update time is dominated by (i) the time required to initialize the data structure of Lemma 5.1 for G^u, and (ii) the time required to maintain $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ for every edge $xy \in E$. By Lemma 5.1, the time for (i) is at most $O(m)$. To bound the time for (ii), note that $\mathrm{c}(xy)$ for any edge $xy \in E$ can be updated in $O(1)$ time as we only inspect the contribution of u towards $\mathrm{c}(xy)$ as discussed before. Lemma 5.1 guarantees that each $\textsc{In}(\cdot)$ or $\textsc{Out}(\cdot)$ is supported in time $O(1)$, which in turn implies that $\mathrm{t}(xy)$ can be updated in $O(1)$ time. As there are at most m edges in G, maintaining $\mathrm{c}(\cdot)$ and $\mathrm{t}(\cdot)$ takes at most $O(m)$. ◀

5.2 Edge deletions

After the deletion of E_{del}, the data structure passes the deletion to *every* G^u, $u \in V$. Let D^u and A^u denote the sets of vertices removed from Desc^u and Anc^u, resp., due to the deletion of E_{del}.

We decrementally maintain Desc^u and Anc^u, and the sets D^u and A^u using the data structure of Lemma 5.1, which is initialized last time G^u was rebuilt due to an insertion centered around u.

To maintain $\mathrm{c}(xy)$ for any edge $xy \in E$, we need to cancel out every vertex $z \notin \{x, y\}$ that contained a path $x \to z \to y$ in G^z before the update but no longer has one. i.e., $x \in \mathrm{Anc}^z \cup \mathrm{A}^z$ and $y \in \mathrm{Desc}^z \cup \mathrm{D}^z$ in G^z and either $x \in \mathrm{A}^z$ or $y \in \mathrm{D}^z$. This suggests that it suffices to subtract $\mathrm{c}(xy)$ by one if x and y fall into one of the following disjoint cases.

1. $x \in \mathrm{A}^z$ and $y \in \mathrm{Desc}^z$, or
2. $x \in \mathrm{A}^z$ and $y \in \mathrm{D}^z$, or
3. $x \in \mathrm{Anc}^z$ and $y \in \mathrm{D}^z$.

For cases (1) and (2) where $x \in \mathrm{A}^z$, we can afford to inspect every outgoing edge $xy \in E^z$ of x with $y \neq z$, and subtract $\mathrm{c}(xy)$ by one if $y \in \mathrm{Desc}^z \cup \mathrm{D}^z$. For case (3) where $y \in \mathrm{D}^z$, we inspect every incoming edge $xy \in E^z$ of y with $x \neq z$, and subtract $\mathrm{c}(xy)$ by one if $x \in \mathrm{Anc}^z$.

To maintain $\mathrm{t}(xy)$ for an edge $xy \in E$, we only need to inspect the updated graphs G^x and G^y. Note that since the graphs are decremental, $\mathrm{t}(xy)$ can only change from 1 to 0. we check whether there is no path $x \to z \to y$ in G^x and G^y passing through a vertex $z \notin \{x, y\}$ by utilizing the data structure of Lemma 5.1: if both $\textsc{In}(y)$ and $\textsc{Out}(x)$ return `False`, we set $\mathrm{t}(xy) \leftarrow 0$.

▶ **Lemma 5.4.** *After each deletion, the data structure maintains the transitive reduction G^t of G in $O(m)$ amortized time, where m is the number of edges in the current graph G.*

Proof. Suppose that a deletion of edges E_{del} has happened.

Correctness. Similar to handling edge insertions. The correctness follows from Lemma 5.2 and the observation that the values $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ for every edge $xy \in E$ are maintained correctly.

Update time. The update time is dominated by (i) the time required to decrementally maintain the data structures of Lemma 5.1 for G^v, $v \in V$, and (ii) the time required to update the values $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ for every edge $xy \in E$.

By Lemma 5.1, it follows that the total time required to maintain (i) over a sequence of m edge deletions is bounded by $O(m)$ for each graph. Therefore, it takes $O(n)$ amortized time to decrementally maintain all graphs.

To bound (ii), we first examine the total time required to maintain $\mathrm{c}(xy)$ over a sequence of m edge deletions in a single graph G^z. Note that in G^z, every vertex $u \in V$ is inspected at most twice since it is deleted at most once from each set Desc^z or Anc^z. During the inspection of u in G^z, the data structure examines each incoming or outgoing edge e of u in $O(1)$ time, and updates $\mathrm{c}(e)$ if necessary. As explained before, the value $\mathrm{c}(e)$ for a single edge e can be updated in $O(1)$ time. Thus, in G^z, the time required for maintaining $\mathrm{c}(xy)$ for every $xy \in E^z$ is bounded by $O\left(\sum_{u \in \mathrm{D}^z} \deg(u) + \sum_{u \in \mathrm{A}^z} \deg(u)\right)$, where D^z and A^z are the vertices that no longer belong to Desc^z and Anc^z, respectively, due to the deletion of E_{del}. Note that, during a sequence of m edge deletions, sets D^z and A^z partition the vertices of G^z, and so the total time for maintaining $\mathrm{c}(xy)$, $xy \in E^z$, in G^z is bounded by $O\left(\sum_{u \in V} \deg(u) + \sum_{u \in V} \deg(u)\right) = O(m)$. We conclude that maintaining $\mathrm{c}(xy)$, $xy \in E$, in all graphs $G^{\cdot}$ takes $O(n)$ amortized time.

Lemma 5.1 guarantees that each in-neighbor or out-neighbor query can also be supported in $O(1)$ time, which in turn implies that $\mathrm{t}(xy)$ for a an edge $xy \in E$ can be updated in $O(1)$ time. As there are at most m edges in the current graph G, updating $\mathrm{t}(xy)$ for all $xy \in E$ costs $O(m)$ after each delete operation. Since the number of delete operations is bounded by the number of edges that appeared in G, we conclude that the total cost to maintain $\mathrm{t}(xy)$ for all $xy \in E$ over all edge deletions is $O(m^2)$, which bounds (ii).

Combining the bounds we obtained for (i) and (ii), we conclude that edge deletions can be supported in $O(m)$ amortized update time. ◀

5.3 Space complexity

It remains to discuss the space complexity of the data structure. Note that explicitly storing all graphs would require $\Omega(n^2 + nm)$ space. In the rest of this section, we sketch how to decrease the space to $O(n^2)$ using a similar approach in [45].

For every edge $xy \in E$, we define a *timestamp* time(xy), attached to xy, denoting its time of insertion into G. We maintain a single explicit adjacency list representation of G, so that the outgoing incident edges $E[v]$ of each v are stored in increasing order by their timestamps. This adjacency list is shared by all the snapshots, and is easily maintained when edges of G are inserted or removed: insertions can only happen at the end of these lists, and deletions can be handled using pointers into this adjacency list.

Let time(v) denote the last time an insertion centered at v happened. Let us order the vertices $V = \{v_1, \ldots, v_n\}$ so that time$(v_i) <$ time(v_j) for all $i < j$, i.e., from the least to most recently updated. By the definition of snapshots, we have

$$G^{v_1} \subseteq G^{v_2} \subseteq \ldots \subseteq G^{v_n} = G.$$

Note that for each i, the edges of G^{v_i} that are not in $G^{v_{i-1}}$ are all incident to v_i and have timestamps larger than timestamps of the edges in $G^{v_{i-1}}$. As a result, one could obtain the adjacency list representation of G^{v_i} by taking the respective adjacency list of G and truncating all the individual lists $E[v]$ at the last edge e with time$(e) \leq$ time(v_i). This idea gives rise to a *virtual* adjacency list of G^{v_i}, which requires only storing time(v_i) to be accessed. One can thus process G^{v_i} by using the global adjacency list for G and ensuring to never iterate through the "suffix" edges in $E[v]$ that have timestamps larger than time(v_i). Using the timestamps, it is also easy to notify the required individual snapshots when an edge deletion in G affects them.

Since all the auxiliary data structures for individual snapshots apart from their adjacency lists used $O(n)$ space, this optimization decreases space to $O(n^2)$.

6 Algebraic Dynamic Transitive Reduction on DAGs

In this section we give algebraic dynamic algorithms for transitive reduction in DAGs (for general digraphs, see the full version of the paper). We reduce the problem to maintaining the inverse of a matrix associated with G and (in the general case) testing some identities involving the elements of the matrix.

We will need the following results on dynamic matrix inverse maintenance.

▶ **Theorem 6.1.** [47] *Let A be an $n \times n$ matrix over a field $\mathbb{F}$ that is invertible at all times. There is a data structure maintaining A^{-1} explicitly subject to row or column updates issued to A in $O(n^2)$ worst-case update time. The data structure is initialized in $O(n^\omega)$ time and uses $O(n^2)$ space.*

▶ **Theorem 6.2.** [47, 48] *Let A be an $n \times n$ matrix over a field $\mathbb{F}$ that is invertible at all times, $a \in (0, 1)$, and Y be a (dynamic) subset of $[n]^2$. There is a data structure maintaining A^{-1} subject to* single-element *updates to A that maintains $A^{-1}_{i,j}$ for all $(i, j) \in Y$ in $O(n^{\omega(1,a,1)-a} + n^{1+a} + |Y|)$ worst-case time per update. The data structure additionally supports:*

1. *square submatrix queries $A^{-1}[I, I]$, for $I \subseteq [n]$, $|I| = n^\delta$ in $O(n^{\omega(\delta,a,\delta)})$ time,*
2. *given $A^{-1}_{i,j}$, adding or removing (i, j) from Y, in $O(1)$ time.*

The data structure can be initialized in $O(n^\omega)$ time and uses $O(n^2)$ space.

Above, $\omega(p, q, r)$ denotes the exponent such that multiplying $n^p \times n^q$ and $n^q \times n^r$ matrices over $\mathbb{F}$ requires $O(n^{\omega(p,q,r)})$ field operations. Moreover $\omega := \omega(1, 1, 1)$.

For DAGs, efficient algebraic fully dynamic transitive reduction algorithms follow from Theorems 6.1 and 6.2 rather easily by applying the reduction of Lemma 4.1. To show that, we now recall how the algebraic dynamic transitive closure data structures for DAGs [20, 33] are obtained.

Identify V with $[n]$. Let $A(G)$ be the standard adjacency matrix of $G = (V, E)$, that is, for any $u, v \in V$, $A(G)_{u,v} = 1$ iff $uv \in E$, and $A(G)_{u,v} = 0$ otherwise. It is well-known that the powers of A encode the numbers of walks between specific endpoints in G. That is, for any $u, v \in V$ and $k \geq 0$, $A(G)^k_{u,v}$ equals the number of $u \to v$ k-edge walks in G. In DAGs, all walks are actually paths. Moreover, we have the following property:

▶ **Lemma 6.3.** *If G is a DAG, then the matrix $I - A(G)$ is invertible. Moreover, $(I - A(G))^{-1}_{u,v}$ equals the number of paths from $u \to v$ in G.*

Proof. Put $A := A(G)$. If G is a DAG, then A^n is a zero matrix since every n-edge walk has to contain a cycle. From that it follows that $(I - A)(I + A + \ldots + A^{n-1}) = I$, i.e., $\sum_{i=0}^{n-1} A^i$ is the inverse of $I - A$. On the other hand, the former matrix clearly encodes path counts of all the possible lengths $0, 1, \ldots, n-1$ in G. ◀

By the above lemma, to check whether a path $u \to v$ exists in a DAG G, it is enough to test whether $(I - A(G))^{-1}_{u,v} \neq 0$. This reduces dynamic transitive closure on G to maintaining the inverse of $I - A(G)$ dynamically. There are two potential problems, though: (1) (unbounded) integers do not form a field (and Theorems 6.1 and 6.2 are stated for a field), (2) the path counts in G may be very large integers of up to $\Theta(n)$ bits, which could lead to an $\widetilde{O}(n)$ bound for performing arithmetic operations while maintaining the path counts. A standard way to address both problems (with high probability $1 - 1/\operatorname{poly}(n)$) is to perform all the counting modulo a sufficiently large random prime number p polynomial in n (see, e.g., [33, Section 3.4] for an analysis). Working over $\mathbb{Z}/p\mathbb{Z}$ solves both problems as arithmetic operations modulo p can be performed in $O(1)$ time on the word RAM.

▶ **Theorem 6.4.** *Let G be a fully dynamic DAG. The transitive reduction of G can be maintained:*
1. *in $O(n^2)$ worst-case time per update if vertex updates are allowed,*
2. *in $O(n^{1.528} + m)$ worst-case time per update if only single-edge updates are supported.*
Both data structures are Monte Carlo randomized and give correct outputs with high probability. They can be initialized in $O(n^\omega)$ time and use $O(n^2)$ space.

Proof. Lemma 4.1 reduces maintaining the set of redundant edges of G to a dynamic transitive closure data structure maintaining reachability for $O(m)$ pairs $Y \subseteq A \times A$, supporting either:
1. vertex-centered updates to both the underlying graph and the set Y, or
2. single-edge updates to the underlying graph and single-element updates to Y.
By our discussion, the former data structure can be obtained by applying Theorem 6.1 to the matrix $I - A(G)$. This way, for vertex updates, one obtains $O(n^2)$ worst-case update time immediately.

The latter data structure is obtained by applying Theorem 6.2 with $a \approx 0.528$ that satisfies $\omega(1, a, 1) = 1 + 2a$ to the matrix $I - A(G)$. Then, a single-edge update is handled in $O(n^{1+a} + m)$ time. Note that inserting a new element to Y in Theorem 6.2 requires computing the corresponding element of the inverse. This requires $O(n^{\omega(0,a,0)}) = O(n^a)$ extra time and is negligible compared to the $O(n^{1+a})$ cost of the element update. ◀

References

1 Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. In *50th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 261, pages 6:1–6:17, 2023. doi:10.4230/LIPICS.ICALP.2023.6.

2 Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *57th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 335–344, 2016. doi:10.1109/FOCS.2016.44.

3 Satabdi Aditya, Bhaskar DasGupta, and Marek Karpinski. Algorithmic perspectives of network transitive reduction problems and their applications to synthesis and analysis of biological networks. *CoRR*, abs/1312.7306, 2013. arXiv:1312.7306.

4 Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972. doi:10.1137/0201008.

5 Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discret. Comput. Geom.*, 9:81–100, 1993. doi:10.1007/BF02189308.

6 Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. doi:10.1145/2344422.2344425.

7 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007. doi:10.1002/RSA.20130.

8 András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 47–55. ACM, 1996. doi:10.1145/237814.237827.

9 András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015. doi:10.1137/070705970.

10 Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. *ACM Trans. Algorithms*, 17(4):29:1–29:51, 2021. doi:10.1145/3469833.

11 Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Decremental strongly connected components and single-source reachability in near-linear time. *SIAM J. Comput.*, 52(on):128–155, 2019. doi:10.1137/20M1312149.

12 Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In *49th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 229, pages 20:1–20:20, 2022. doi:10.4230/LIPICS.ICALP.2022.20.

13 Dragan Bonaki, Maximilian R. Odenbrett, Anton Wijs, Willem P. A. Ligtenberg, and Peter A. J. Hilbers. Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. *BMC Bioinform.*, 13:281, 2012. doi:10.1186/1471-2105-13-281.

14 Ruoxu Cen, Yu Cheng, Debmalya Panigrahi, and Kevin Sun. Sparsification of directed graphs via cut balance. In *48th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 198, pages 45:1–45:21, 2021. doi:10.4230/LIPICS.ICALP.2021.45.

15 Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008. doi:10.1007/S00453-007-9062-1.

16 Joseph Cheriyan and Ramakrishna Thurimella. Approximating minimum-size k-connected spanning subgraphs via matching. *SIAM J. Comput.*, 30(2):528–560, 2000. doi:10.1137/S009753979833920X.

17 James R. Clough, Jamie Gollings, Tamar V. Loach, and Tim S. Evans. Transitive reduction of citation networks. *J. Complex Networks*, 3(2):189–203, 2015. doi:10.1093/COMNET/CNU039.

18 Michael B. Cohen, Jonathan A. Kelner, John Peebles, Richard Peng, Aaron Sidford, and Adrian Vladu. Faster algorithms for computing the stationary distribution, simulating random walks, and more. In *57th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 583–592, 2016. doi:10.1109/FOCS.2016.69.

19 Don Coppersmith and Shmuel Winograd. On the asymptotic complexity of matrix multiplication. *SIAM J. Comput.*, 11(3):472–492, 1982. doi:10.1137/0211038.

20 Camil Demetrescu and Giuseppe F. Italiano. Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n^2 barrier. *J. ACM*, 52(2):147–156, 2005. doi:10.1145/1059513.1059514.

21 Vincent Dubois and Cécile Bothorel. Transitive reduction for social network analysis and visualization. In *IEEE WIC ACM International Conference on Web Intelligence (WI)*, pages 128–131, 2005. doi:10.1109/WI.2005.152.

22 Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *51st Annual ACM Symposium on Theory of Computing (STOC)*, pages 377–388, 2019. doi:10.1145/3313276.3316381.

23 Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. *SIAM J. Comput.*, 48(4):1196–1223, 2019. doi:10.1137/16M1091666.

24 Loukas Georgiadis, Giuseppe F. Italiano, Aikaterini Karanasiou, Charis Papadopoulos, and Nikos Parotsidis. Sparse subgraphs for 2-connectivity in directed graphs. In *15th European Symposium on Algorithms (ESA)*, volume 9685, pages 150–166, 2016. doi:10.1007/978-3-319-38851-9_11.

25 Phillip B. Gibbons, Richard M. Karp, Vijaya Ramachandran, Danny Soroker, and Robert Endre Tarjan. Transitive compaction in parallel via branchings. *J. Algorithms*, 12(1):110–125, 1991. doi:10.1016/0196-6774(91)90026-U.

26 Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine A. Yelick, and Aydin Buluç. Parallel string graph construction and transitive reduction for de novo genome assembly. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 517–526, 2021. doi:10.1109/IPDPS49936.2021.00060.

27 Xiaofeng Han, Pierre Kelsen, Vijaya Ramachandran, and Robert Endre Tarjan. Computing minimal spanning subgraphs in linear time. *SIAM J. Comput.*, 24(6):1332–1358, 1995. doi:10.1137/S0097539791224509.

28 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

29 Shuquan Hou, Xinyi Huang, Joseph K. Liu, Jin Li, and Li Xu. Universal designated verifier transitive signatures for graph-based big data. *Information Sciences*, 318:144–156, 2015. Security, Privacy and trust in network-based Big Data. doi:10.1016/j.ins.2015.02.033.

30 Giuseppe F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988. doi:10.1016/0020-0190(88)90136-6.

31 Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: scaling reachability computation on large graphs. In *ACM International Conference on Management of Data (SIGMOD)*, pages 169–180, 2012. doi:10.1145/2213836.2213856.

32 Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 81–91, 1999. doi:10.1109/SFFCS.1999.814580.

33 Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002. doi:10.1006/JCSS.2002.1883.

34 Steffen Klamt, Robert J. Flassig, and Kai Sundmacher. TRANSWESD: inferring cellular networks with transitive reduction. *Bioinform.*, 26(17):2160–2168, 2010. doi:10.1093/BIOINFORMATICS/BTQ342.

35 Bundit Laekhanukit, Shayan Oveis Gharan, and Mohit Singh. A rounding by sampling approach to the minimum size k-arc connected subgraph problem. In *39th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 7391, pages 606–616, 2012. doi:10.1007/978-3-642-31594-7_51.

36 De Meng. Transitive reduction approach to large-scale parallel machine rescheduling problem with controllable processing times, precedence constraints and random machine breakdown. *IEEE Access*, 11:7727–7738, 2023. doi:10.1109/ACCESS.2023.3238639.

37 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992. doi:10.1007/BF01758778.

38 Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. *SIGPLAN Not.*, 28(12):1–11, December 1993. doi:10.1145/174267.174268.

39 David Peleg and Alejandro A. Schäffer. Graph spanners. *J. Graph Theory*, 13(1):99–116, 1989. doi:10.1002/JGT.3190130114.

40 Andrea Pinna, Sandra Heise, Robert J. Flassig, Alberto de la Fuente, and Steffen Klamt. Reconstruction of large-scale regulatory networks based on perturbation graphs and transitive reduction: improved methods and their evaluation. *BMC Syst. Biol.*, 7:73, 2013. doi:10.1186/1752-0509-7-73.

41 Johannes A. La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *13th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 314, pages 106–120, 1987. doi:10.1007/3-540-19422-3_9.

42 Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1):6:1–6:16, 2008. doi:10.1145/1328911.1328917.

43 Liam Roditty, Mikkel Thorup, and Uri Zwick. Roundtrip spanners and roundtrip routing in directed graphs. *ACM Trans. Algorithms*, 4(3):29:1–29:17, 2008. doi:10.1145/1367064.1367069.

44 Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

45 Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016. doi:10.1137/13093618X.

46 Sushant Sachdeva, Anvith Thudi, and Yibin Zhao. Better sparsifiers for directed eulerian graphs. In *51st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 297, pages 119:1–119:20, 2024. doi:10.4230/LIPICS.ICALP.2024.119.

47 Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 509–517, 2004. doi:10.1109/FOCS.2004.25.

48 Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *11th International Computing and Combinatorics Conference (COCOON)*, volume 3595, pages 461–470, 2005. doi:10.1007/11533719_47.

49 Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011. doi:10.1137/080734029.

50 Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011. doi:10.1137/08074489X.

51 Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969. doi:10.1007/BF02165411.

52 Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991. doi:10.1137/0220006.

53 Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 456–480, 2019. doi:10.1109/FOCS.2019.00036.

54 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 887–898, 2012. doi:10.1145/2213977.2214056.

55 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *35th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835, 2024. doi:10.1137/1.9781611977912.134.

56 Adrian Vetta. Approximating the minimum strongly connected subgraph via a matching lower bound. In *12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 417–426, 2001. URL: `http://dl.acm.org/citation.cfm?id=365411.365493`.

57 Min Xu, Mark D. Hill, and Rastislav Bodík. A regulated transitive reduction (RTR) for longer memory race recording. In *12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, 2006. `doi:10.1145/1168857.1168865`.