Fully Dynamic Algorithms for Transitive Reduction

Gramoz Goranci[1], Adam Karczmarz*[2], Ali Momeni[3], and Nikos Parotsidis[4]

[1]Faculty of Computer Science, University of Vienna, Austria
[2]University of Warsaw
[3]Faculty of Computer Science, UniVie Doctoral School Computer Science DoCS, University of Vienna, Austria
[4]Google Research, 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.

*Supported by the National Science Centre (NCN) grant no. 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 [AGU72; ADDJS93; BK96; CT00; ST11; SS11]. 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 to Boolean Matrix Multiplication, the best known efficient algorithm runs in $O(n^\omega)$ time, where $\omega < 2.371552$ [Str69; CW82; Vas12; VXXZ24]. In sparse graphs, transitive closure can be trivially computed in $O(nm)$ time[1].

In their seminal work, Aho, Garey, and Ullman [AGU72] 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, [AGU72] 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 [AGU72]. 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 [GKRST91; HKRT95]. 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 [AGU72] 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

[1] Interestingly, shaving a logarithmic factor is possible here [Cha08].

[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.

the reconstruction of biological networks [BOWLH12; GSEOYB21] and other types of networks (e.g., [PHFFK13; KFS10; ADK13]), in code analysis and debugging [XHB06; Net93], network analysis and visualization for social networks [DB05; CGLE15], signature verification [HHLLX15], serving reachability queries in database systems [JRDY12], large-scale parallel machine rescheduling problems [Men23], and many more. In some of these applications (e.g., [BOWLH12; GSEOYB21]), 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 time $O\left(m \cdot \min(n^\omega, nm)\right)$, or in other words, the algorithm has $O\left(\min(n^\omega, nm)\right)$ 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 [PL87] 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,*

[4]i.e. that can be removed from the graph without affecting the transitive closure.

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 5.1 and Section 5.2.

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*

(1) in $O(n^2)$ worst-case update time for vertex updates, and

(2) 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*

(1) in $O(n^2)$ worst-case update time for vertex updates, and

(2) 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 [PL87]. 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 [HKRT95; NI92], shortest paths [ADDJS93; PS89; BS07], cut sparsifiers [BK96; BK15; FHHP19], spectral sparsifiers [SS11; ST11], and many more. For directed graphs, applications of sparsification have been studied for reachability [AGU72], strong connectivity [HKRT95; Vet01], shortest paths [Kin99; RTZ08], k-connectivity [GIKPP16; LGS12; CT00], cut sparsifiers [CCPS21], spectral sparsifiers for Eulerian digraphs [CKPPSV16; STZ24], 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 [HLT01], shortest paths [BKS12; FG19; BFH21], cut and spectral sparsifiers [ADKKP16; BBGNSSS22], and k-edge-connectivity [AKLPRT23].

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 combinatorial data structures for DAGs and general graphs in Section 5, followed by the algebraic data structures in Section 6.

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 [AGU72]. 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 ([AGU72]). *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 [AGU72]. 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 [Rod08; RZ16] and algebraic [BNS19; San04] data structures known. As mentioned before, computing transitive reduction is closely related to computing the transitive closure [AGU72]. 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 [Rod08; San04] 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 [BNS19; RZ16; San04].

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 [AGU72] (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 [Rod08; San04], 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]

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 [San04; San05] 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 [VXXZ24] 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 [BNS19]. 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 [RZ08], 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 [Ita88], 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 [RZ08; RZ16]: 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 [RZ08; RZ16], 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 path. Using this technique, if the count is zero for some edge uv, then uv is not redundant. This allows

[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 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 [RZ08; RZ16], 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.

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 [GKRST91].

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 [Ita88] breaks for general graphs, though. One solution here would be to adapt the near-linear decremental single-source reachability for general graphs [BGW19]. However, that data structure is randomized, slower by a few log factors, and much more complex. Instead, as in [RZ08; RZ16], 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 [RZ16] 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 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

[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).

be efficiently addressed within the amortized bound of $O(m + n \log n)$ which matches that of the fully dynamic reachability data structure of [RZ16]. 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 Section 5.2.

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 [BNS19; San04]. 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* [San04] (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 (Theorem 6.5) 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 Theorem 6.8, item (1)).

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 [BNS19; San04] (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 [UY91] 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 Section 6.2.

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 $Y = \{(x_i, y_i) : i = 1, \dots, k\} \subseteq V \times V$. Suppose there exists a data structure maintaining whether paths $x_i \to y_i$ exist in G for $i = 1, \dots, k$ 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. TOPROVE 0 □

5 Combinatorial Dynamic Transitive Reduction

We start by explaining the data structure for DAGs in Section 5.1, and then present the general case in Section 5.2.

5.1 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.

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 [Ita88], summarized in the lemma below. See Appendix A 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:*

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

(2) *Vertex y has an in-neighbor $z \in \mathrm{Desc}^x \setminus x$ in G^x.*

(3) *Vertex x has an out-neighbor $z \in \mathrm{Anc}^y \setminus y$ in G^y.*

Proof. TOPROVE 1 □

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

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

Note that for a redundant edge xy in G, a vertex z contributes towards $\mathrm{c}(xy)$ 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 $\mathrm{t}(xy) = 1$. 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 $\mathrm{c}(xy) = 0$ and $\mathrm{t}(xy) = 0$.*

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

5.1.1 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 $\mathrm{c}(xy)$ for an edge $xy \in E$, we only need to examine the contribution of u in $\mathrm{c}(xy)$ 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 $\mathrm{c}(xy)$. 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:

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

(b) $y = u$ 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. TOPROVE 2 □

5.1.2 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. TOPROVE 3 □

5.1.3 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 [RZ16].

For every edge $xy \in E$, we define a *timestamp* $\mathrm{time}(xy)$, 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 $\mathrm{time}(v)$ denote the last time an insertion centered at v happened. Let us order the vertices $V = \{v_1, \ldots, v_n\}$ so that $\mathrm{time}(v_i) < \mathrm{time}(v_j)$ 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 $\mathrm{time}(e) \leq \mathrm{time}(v_i)$. This idea gives rise to a *virtual* adjacency list of G^{v_i}, which requires only storing $\mathrm{time}(v_i)$ 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 $\mathrm{time}(v_i)$. 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 $O(n^2)$.

5.2 Combinatorial Dynamic Transitive Reduction on General Graphs

In this section, we explain our data structure when G is a general graph, as summarized in Theorem 1.2 below.

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.*

Analogous to Section 5.1, we maintain the reachability information on a decremental 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.

The main difference here from DAGs is the existence of strongly connected components (SCCs) with arbitrary size. We define the *condensation* of G to be the DAG obtained by contracting each SCC of G. Our goal is to maintain the *inter-SCC* and *intra-SCC* edges belonging to the transitive reduction G^t of G.

To maintain the inter-SCC edges of G^t, we adapt our data structure on DAGs utilized on the condensation of G, equipped to address two challenges that did not arise during the designing of the data structure of Section 5.1: (i) each update could cause vertex splits or vertex merges in the condensation of G, and (ii) the condensation of G consists of parallel edges. To address this challenges, we use an extended version of the decremental data structure of [RZ16] explained in Appendix B and summarized in the lemma below. This data structure maintains SCCs of G^u for each $u \in V$ and the set of parallel edges between the SCCs in G, which will be used to address (i) and (ii), respectively.

Lemma 5.5. *Given a directed graph $G = (V, E)$ and the subgraphs G^r for every $r \in V$ 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 that maintains the SCCs and the set of parallel inter-SCC edges of all graphs in $O(m + n \log n)$ amortized update time, where m is the number of edges in the current graph G.*

Additionally, for all $r \in V$, the data structure maintains the sets D^r and A^r of vertices removed from Desc^r and Anc^r, respectively, as a consequence of the last update and supports the following operations in time $O(1)$, where X, Y, R are the SCCs containing x, y, r, respectively:

- $\textsc{In}(y, r)$: *return* `True` *if, in G^r, $Y \neq R$ and Y has an in-neighbor from $\mathrm{Desc}^r \setminus R$, and* `False` *otherwise.*
- $\textsc{Out}(x, r)$: *return* `True` *if, in G^r, $X \neq R$ and X has an out-neighbor to $\mathrm{Desc}^r \setminus R$, and* `False` *otherwise.*

Lastly, the data structure maintains the set S^r of the SCCs in G^r such that $\textsc{In}(y, r)$ or $\textsc{Out}(x, r)$ has changed from `True` *to* `False` *due to the last deletion.*

We denote by $\mathrm{F}[X, Y]$ the set of parallel edges in G between the SCCs X, Y maintained by Lemma 5.5. The lemma below shows how maintaining Desc^u, Anc^u, and the parallel edges is 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 5.6. *Let $xy \in E$ be an inter-SCC edge of G, and for a vertex $z \in V$, let X, Y, Z be the SCCs containing x, y, z in G^z, respectively. Then xy is redundant in G iff one of the following holds.*

1. *Vertex z exists with $Z \notin \{X, Y\}$ such that $xy \in E^z$, $x \in \mathrm{Anc}^z$, and $y \in \mathrm{Desc}^z$ in G^z.*
2. *Vertex z exists with $Z = X$ in G^z, such that Y has an in-neighbor from $\mathrm{Desc}^z \setminus Z$ in G^z.*
3. *Vertex z exists with $Z = Y$ in G^z, such that X has an out-neighbor to $\mathrm{Anc}^z \setminus Z$ in G^z.*
4. *There are parallel edges between the SCCs X and Y in G.*

Proof. TOPROVE 4 □

Note that when no parallel inter-SCC edge exists, an inter-SCC edge xy belongs to G^t iff it is not redundant, as the latter guarantees that there is no simple path $x \to z \to y$ where z is not belonging to the SCCs containing x, y. On the other hand, the existence of parallel edges does not change the reachability information nor the size of G^t: if an inter-SCC edge xy belongs to G^t, then all other parallel edges between X and Y cannot belong to G^t. This motivates us to "ignore" parallel inter-SCC edges except one of them in our data structure: we *mark* the front edge in $\mathrm{F}[X, Y]$ and simply assume that other edges does not belong to G^t. Therefore, if an inter-SCC edge is not marked, then it does not belong to G^t, and if it is marked and not redundant, then it belongs to G^t.

To incorporate Lemma 5.6 in our data structure, we define $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ for every inter-SCC edge xy as follows, where X, Y, Z are the SCCs containing x, y, z, respectively, in G^z:

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

Note that for a redundant edge xy in G, a vertex z contributes towards $\mathrm{c}(xy)$ iff $xy \in E^z$ as an inter-SCC edge. If there is no such z, then xy has become redundant due the last insertion being centered around x or y, which in turn implies that $\mathrm{t}(xy) = 1$. We conclude the following invariant.

Invariant 5.2. *An inter-SCC edge $xy \in E$ belongs to the transitive reduction G^t iff xy is a marked parallel edge with $c(xy) = 0$ and $t(xy) = 0$.*

In Sections 5.2.1 and 5.2.2, we explain how we efficiently maintain $\mathrm{c}(xy)$ and $\mathrm{t}(xy)$ for every inter-SCC edge $xy \in E$. In the following, we explain how to maintain the intra-SCC edges belonging to G^t.

For intra-SCC edges, we use the following lemma.

Lemma 5.7 ([GKRST91]). *Given an m-edge n-vertex strongly connected graph G, there is an algorithm to compute a minimal strongly connected subgraph of G in $O(m + n \log n)$ worst-case time.*

After each update, we run the algorithm of Lemma 5.7 on all SCCs of G maintained by the data structure of Lemma 5.5. Since this takes $O(m + n \log n)$ time, this does not affect the update time of our data structure.

In the rest of this section, we focus on maintaining inter-SCC edges as the maintenance of intra-SCC edges is clear from the discussion above. Each edge will be assumed to be an inter-SCC edge, unless explicitly expressed otherwise.

5.2.1 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*.

The data structure first computes the sets Desc^u and Anc^u, as well as the sets C^u and B^u of the vertices added to Desc^u and Anc^u, respectively, due to the insertion of E_u. It computes the SCCs of G^u while listing the edges entering each SCC $Z \neq U$ from $\mathrm{Desc}^u \setminus U$ and the edges leaving the SCC to $\mathrm{Anc}^u \setminus U$, where U is the SCC in G^u containing u. By inspecting E^u, the data structure computes the set $\mathrm{F}[X, Y]$ of parallel edges between each pair X, Y of SCCs, thus maintaining the marked edges. This takes $O(m + n \log n)$ time and reinitializes the data structure of Lemma 5.5 for G^u.

To maintain $\mathrm{c}(xy)$ for every inter-SCC edge $xy \in E$, we only need to examine the contribution of u in $\mathrm{c}(xy)$ as only G^u has changed. By definition, every inter-SCC edge xy touching the SCC U, i.e., with $x \in U$ or $y \in U$, does not contribute in $\mathrm{c}(xy)$. Note that E^u consists of the edges before the update and the newly added edges, and so U could only grow after the update. This 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 $y \in \mathrm{Desc}^u$).

(3) Suppose that xy is a newly edge touched by U. i.e., xy touching U as a result of the growth of U after the update. Then, $\mathrm{c}(xy)$ will decrease by one only if the update makes x could reach y through u before the update ($x \in \mathrm{Anc}^u \setminus \mathrm{B}^u$ and $y \in \mathrm{Desc}^u \setminus \mathrm{C}^u$).

Cases (1) and (2) are handled by inspecting the inter-SCC edges touching $\mathrm{B}^u \cup \mathrm{C}^u$. To prevent double-counting, a similar partitioning in Section 5.1.2 can be used. For case (3), we can afford to inspect all inter-SCC edges touching the vertices added to U due to the update.

To maintain $\mathrm{t}(xy)$ for every inter-SCC edge $xy \in E$, we distinguish the two following cases. Since only G^u changes, by definition, $\mathrm{t}(xy)$ could be affected only if xy touches U. As the update on G^u adds more edges to it, $\mathrm{t}(xy)$ 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:

(a) $x \in U$ and Y has an in-neighbor $w \notin U$ in G^u reachable from u (i.e., if $\textsc{In}(y, x)$ reports `True`), or

(b) $y \in U$ and X has an out-neighbor $w \notin U$ in G^u that can reach Y (i.e., if $\textsc{Out}(x, y)$ reports `True`).

Note that both cases above try to insure the existence of a path $x \to w \to y$ when $x \in U$ or $y \in U$.

Lemma 5.8. *After each insertion, the data structure maintains the transitive reduction G^t of G in $O(m + n \log n)$ worst-case time, where m is the number of edges in the current graph G.*

Proof. TOPROVE 5 □

5.2.2 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, respectively, 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 5.5, which is initialized last time G^u was rebuilt due to an insertion centered around u. Since the data structure maintains the set $\mathrm{F}[X, Y]$ of parallel edges between each pair X, Y of SCCs, the marked inter-SCC edges are automatically maintained.

To maintain $\mathrm{c}(xy)$ for any inter-SCC 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 $y \in \mathrm{D}^z$. To avoid double-counting u for the edges with both endpoints in $\mathrm{A}^u \cup \mathrm{D}^u$, we use a similar approach explained in Section 5.1.2: partition $\mathrm{A}^u \cup \mathrm{D}^u$ into three disjoint sets and argue how to handle each. Contrary to DAGs, there may be new inter-SCC edges form in G^u due to SCC splits, which are given by the data structure of Lemma 5.5. For each such edge xy, we can check in constant time if $x, y \notin U$, and if they satisfy the conditions of $\mathrm{c}(xy)$, we increase $\mathrm{c}(xy)$ by one.

To maintain $\mathrm{t}(xy)$ for an inter-SCC edge $xy \in E$, we distinguish the following three cases. Let X, Y, U be the SCCs containing x, y, u in G^u, respectively.

(a) Vertex x or y has detached from U due to the update. In this case, if $\textsc{In}(u, y) = \texttt{True}$ or $\textsc{Out}(u, x) = \texttt{True}$ in G^u before the update, we set $\mathrm{t}(xy) \leftarrow \mathrm{t}(xy) - 1$.

(b) Vertices x and y has not detached from U, but there is no longer a $x \to y$ path passing through an SCC different than X, Y in G^u. In this case, we only need to inspect the edges touching the SCCs in S^u: if $\textsc{In}(u, y) = \texttt{True}$ or $\textsc{Out}(u, x) = \texttt{True}$ in G^u before the update, we set $\mathrm{t}(xy) \leftarrow \mathrm{t}(xy) - 1$.

(c) If xy is a new inter-SCC edge in G^u. In this case, we only need to inspect the new inter-SCC edges touching U after the update, and if $\textsc{In}(u, y) = \texttt{True}$ or $\textsc{Out}(u, x) = \texttt{True}$, we set $\mathrm{t}(xy) \leftarrow \mathrm{t}(xy) + 1$.

Lemma 5.9. *After each deletion, the data structure maintains the transitive reduction G^t of G in $O(m + n \log n)$ amortized update time, where m is the number of edges in the current graph G.*

Proof. TOPROVE 6 □

5.2.3 Space complexity

A similar encoding explained in Section 5.1.3 results in $O(n^2)$ space.

6 Algebraic Dynamic Transitive Reduction

We start by explaining the data structure for DAGs in Section 6.1, and then present the general case in Section 6.2.

6.1 Algebraic Dynamic Transitive Reduction on DAGs

In this section we give algebraic dynamic algorithms for transitive reduction in DAGs. 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. [San04] *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. [San04; San05] *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 [DI05; KS02] 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. TOPROVE 7 ☐

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., [KS02, 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. TOPROVE 8 ☐

6.2 Algebraic Dynamic Transitive Reduction on General Graphs

In this section we give algebraic dynamic algorithms for general digraphs.

In general graphs, simple path counting fails since there can be infinite numbers of walks between vertices. This is why the algebraic dynamic transitive closure data structures dealing with general graphs [San04] require more subtle arguments. For maintaining the transitive reduction, we will rely on those as well. Identifying redundant edges – specifically groups of parallel redundant inter-SCC edges – will also turn out more challenging. In fact, the obtained data structure for single-edge updates will not be as efficient as the corresponding data structure for DAGs (for the current fast matrix multiplication exponents).

In this section, we will assume that $G = (V,E)$ has *a self loop on every vertex*, i.e., $uu \in E$ for all $u \in V$, even though loops are clearly redundant from the point of view of reachability. For this reason, we also assume the updates only add or delete non-loop edges. When reporting the transitive reduction, we do not report the n self-loop edges that are only helpful internally.

6.2.1 The data structure

For general graphs we will identify redundant intra-SCC and inter-SCC edges separately. Upon update issued to G, the first step is always to compute the strongly-connected components of G from scratch in $O(m)$ time. This yields the partition of edges E into inter- and intra-SCC, and also sets of *parallel* inter-SCC edges, i.e., those that connect the same pair of SCCs.

The intra-SCC edges can be handled in $O(m + n \log n)$ time worst-case time per update using Lemma 5.7 as was done in Section 5.2. In the following, we focus on handling the inter-SCC edges.

Fix a group F of k parallel inter-SCC edges $\{u_iv_i : i = 1, \ldots, k\}$, where $\{u_1, \ldots, u_k\}$ are in the same SCC R of G and $\{v_1, \ldots, v_k\}$ are in the same SCC T of G, and $R \neq T$. Recall that F is redundant if *all* edges in F can be removed without changing the transitive closure of G. If F is non-redundant, we need to keep exactly one, arbitrarily chosen edge of F in the transitive reduction: in such a case, after contracting the SCCs and removing parallel edges, F corresponds to a redundant edge of the obtained DAG. Therefore, given F, we only need to decide whether this group is redundant.

Let r, t be arbitrarily chosen vertices of R, T respectively. Note that F is redundant if and only if there exists a path $r \to t$ going through a vertex in some strongly connected component $Y \notin \{R, T\}$ of G. Our strategy will be to express this property of F algebraically. To this end, we need to introduce some more notions.

Let $X = \{x_{u,v} : u, v \in V\}$ be a set of n^2 formal variables associated with pairs of vertices of G. Fix some finite $\mathbb{F}$. Denote by $\mathbb{F}(X)$ the field of multivariate rational functions with coefficients from $\mathbb{F}$ and variables from X. Following [San04], let us define a *symbolic adjacency matrix* $\tilde{A}(G) \in (\mathbb{F}(X))^{n \times n}$ of G to be a matrix such that

$$\tilde{A}(G)_{u,v} = \begin{cases} x_{u,v} & \text{if } uv \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The self loops in G guarantee that $\tilde{A}(G)$ is invertible over $\mathbb{F}(X)$, i.e., $\det(\tilde{A}(G)) \neq 0$ [San04]. In Section 6.2.3 we are going to prove the following theorem.

Theorem 6.5. *Let $F = \{u_iv_i : i = 1, \ldots, k\}$ be a group of parallel inter-SCC edges between SCCs R, T of G. Let r, t be arbitrary vertices of R, T, resp. The group F is redundant iff:*

$$\tilde{A}(G)^{-1}_{r,t} \not\equiv -\sum_{i=1}^{k} (-1)^{u_i+v_i} \cdot x_{u_i,v_i} \cdot \tilde{A}(G)^{-1}_{r,u_i} \cdot \tilde{A}(G)^{-1}_{v_i,t}.$$

Let us now show how Theorem 6.5 can be used to obtain a dynamic transitive reduction data structure. It is enough to "maintain" the inverse of the matrix $\tilde{A}(G)$ and, after each update, check appropriate polynomial identities on some m entries of the inverse. The inverse's entries are rational functions, so maintaining them would be too slow. Instead, we maintain the entries' evaluations for some uniformly random substitution $\bar{X} : X \to \mathbb{Z}/p\mathbb{Z}$, where $\bar{X} = \{\bar{x}_{u,v} : u, v \in V\}$, and $\mathbb{F} = \mathbb{Z}/p\mathbb{Z}$, where p is a prime number $p = \Theta(n^{3+c})$, $c > 0$.

Lemma 6.6. *Let $\mathbb{F}$ and $A \in \mathbb{F}^{n \times n}$ be a obtained as described above. Then, with probability at least $1 - O(1/n^{2+c})$, A is invertible and F (as defined in Theorem 6.5) is redundant iff:*

$$A^{-1}_{r,t} + \sum_{i=1}^{k} (-1)^{u_i+v_i} \cdot \bar{x}_{u_i,v_i} \cdot A^{-1}_{r,u_i} \cdot A^{-1}_{v_i,t} \neq 0.$$

Proof. TOPROVE 9 □

As G has at most n^2 edges, by the union bound we get that the identity in Lemma 6.6 can be used to correctly classify the inter-SCC edges of G as either redundant or non-redundant with probability at least $1 - O(1/n^c)$. The data structure simply maintains (a part of) the inverse of A and checks the identities of Lemma 6.6 after each update. *Assuming all the needed elements of* A^{-1} *are computed*, testing whether the group F is redundant takes $O(|F|)$ time. As a result, through all the (groups of) edges of G, such a check requires $O(m)$ time. The elements of A^{-1} are never revealed to the adversary, so a single variable substitution picked at the beginning can be used over many updates. By amplifying the constant c, we can guarantee high-probability correctness over a polynomial number of updates to G.

6.2.2 Maintaining the required elements of the inverse A^{-1}.

If we want to support vertex updates changing all edges incident to a single vertex, then we can maintain the entire A^{-1} in $O(n^2)$ worst-case time per update using Theorem 6.1. In such a case, the worst-case update time of our dynamic transitive reduction data structure is $O(n^2)$.

Let us now consider more interesting single-edge updates when G is not dense. Recall that in the acyclic case, it was sufficient to maintain m entries of the inverse corresponding precisely to the edges of G. In the general case, by Lemma 6.6, deciding a group of k parallel inter-SCC edges requires inspecting $2k$ elements of the inverse. However, some of them are not in a 1-1 correspondence with the parallel edges of that group; they depend on the "rooting" (r, t) (of our choice) of the SCCs R and T, respectively. Unfortunately, the groups of parallel edges can change quite dramatically and unpredictably upon updates, e.g., if a group splits as a result of an edge deletion, the chosen rootings might not be helpful at deciding whether the groups after the split are redundant.

We nevertheless obtain a bound close to that we got for DAGs by applying a heavy/light distinction to the SCCs and exploiting randomization further. Let $\delta \in [0, 1]$ be a parameter to be fixed later. We call an SCC *heavy* if it has at least n^δ vertices, and *light* otherwise.

A well-known fact [UY91] says that if we sample a set S of $\Theta(n^{1-\delta} \log n)$ random vertices of G, then any subset of V of size at least n^δ (chosen independently of S) will contain a vertex of S with high probability (dependent on the constant hidden in the Θ notation). In particular, one can guarantee that S *hits* (w.h.p.) every out of poly$\,(n)$ subsets of V chosen independently of S. In our data structure, we sample one such *hitting set* S of size $\Theta(n^{1-\delta} \log n)$ so that it hits all the heavy SCCs that ever arise with high probability. For that, we will never leak S to the adversary (unless the data structure errs, which happens will low probability) so that S remains independent of the current structure of the SCCs. Next, we employ the dynamic matrix inverse data structure $\mathcal{D}$ of Theorem 6.2. We define the "set of interest" Y in that data structure, to contain at all times:

1. $(S \times V) \cup (V \times S)$,
2. (u, v) for all $uv \in E$,
3. $B \times B$ for every *light* SCC B of G.

Note that the size of Y is at most

$$m + 2n|S| + \sum_{\substack{B \subseteq V \\ B \text{ is a light SCC}}} |B|^2 \leq m + \widetilde{O}(n^{2-\delta}) + n^\delta \left(\sum_{\substack{B \subseteq V \\ B \text{ is a light SCC}}} |B| \right) \leq m + \widetilde{O}(n^{2-\delta}) + n^{1+\delta}.$$

Lemma 6.7. *After a single-edge update issued to G, the data structure $\mathcal{D}$ can be updated in $\widetilde{O}(n^{\omega(1,a,1)-a} + n^{1+a} + n^{1-\delta+\omega(\delta,a,\delta)} + m + n^{2-\delta})$ worst-case time.*

Proof. TOPROVE 10 □

Given the elements Y of A^{-1}, let us now prove that we can test the identities in Lemma 6.6 in $O(m)$ time. Consider a group F of parallel inter-SCC edges $\{u_1v_1, \ldots, u_kv_k\}$ so that $u_i \in R$ and $v_i \in T$, where R, T are distinct SCCs of G. We now fix the "rooting" of R and T as follows. If R is heavy, then we pick $r \in R \cap S$, and otherwise, we put $r = u_1$. Similarly, if T is heavy, we pick $t \in T \cap S$, and otherwise, we put $t = v_1$. This way, we have $(r, u) \in Y$ for any $u \in R$, since either R is light and $R \times R \subseteq Y$, or R is heavy and $\{r\} \times R \subseteq S \times V \subseteq Y$. Similarly, one can argue that $(v, t) \in Y$ for any $v \in T$. It follows that all the elements of the form A^{-1}_{r,u_i} or $A^{-1}_{v_i,t}$ required by Lemma 6.6 can be accessed in $O(1)$ time. And so can be the element $A^{-1}_{r,t}$ since $(r, t) \in (S \times V) \cup (V \times S)$ if either S or T is heavy, and we have $(r, s) = (u_1, v_1)$ otherwise, which implies $rs \in E$ and thus $(r, s) \in Y$ as well. We obtain the following:

Theorem 6.8. *Let G be fully dynamic. The transitive reduction of G can be maintained:*

(1) in $O(n^2)$ worst-case time per update if vertex updates are allowed,

(2) for any $a, \delta \in [0, 1]$, in $O(n^{\omega(1,a,1)-a} + n^{1+a} + n^{1-\delta+\omega(\delta,a,\delta)} + n^{2-\delta} + m)$ worst-case time per update if only single-edge updates are supported. In particular, for $a = 0.4345$ and $\delta = 0.415$, this update bound is $O(n^{1.585} + m)$.[8]

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.

6.2.3 Proof of Theorem 6.5

In this section we prove Theorem 6.5 by studying cycle covers; this approach has also been used by [San04].

Let a *cycle cover* of G be a collection of vertex-disjoint simple cycles $\mathcal{C} = \{C_1, \ldots, C_k\}$ such that $C_i \subseteq G$ and $\sum_{i=1}^{k} |C_i| = n$. Denote by $K(G)$ the set of cycle covers of G. Any simple cycle $C \subseteq G$, $C = v_1v_2 \ldots v_\ell$ has an associated monomial

$$\mu(C) = -\prod_{i=1}^{\ell} -x_{v_i,v_{i+1}} = (-1)^{\ell+1} \prod_{i=1}^{\ell} x_{v_i,v_{i+1}},$$

where $v_{\ell+1} := v_1$. For a cycle cover $\mathcal{C} \in K(G)$, we define $\mu(\mathcal{C}) = \prod_{C \in \mathcal{C}} \mu(C)$.

Now, by the Leibniz formula for the determinant, we have:

Fact 6.2.1. $\det(\tilde{A}(G)) = \sum_{\mathcal{C} \in K(G)} \mu(\mathcal{C})$.

As a result, $\tilde{A}(G)$ is invertible, since the polynomial $\det(\tilde{A}(G))$ contains the monomial $\prod_{u \in V} x_{u,u}$ corresponding to the trivial cycle cover of G that consists of self-loops exclusively.

Lemma 6.9. *Let $S_1, \ldots, S_s \subseteq V$ be the strongly connected components of G. Then:*

$$\det(\tilde{A}(G)) = \prod_{i=1}^{s} \det(\tilde{A}(G[S_i])).$$

[8]This choice of parameters a, d can be obtained using the online complexity term balancer [Bra].

Proof. TOPROVE 11 □

Corollary 6.10. *Let $U \subseteq V$ be such that each SCC of G is fully contained in either U or in $V \setminus U$. Then,*

$$\det(\tilde{A}(G)) = \det(\tilde{A}(G[U])) \cdot \det(\tilde{A}(G[V \setminus U])).$$

For $u, v \in V(G)$, let $\tilde{A}^{v,u}(G)$ be obtained from $\tilde{A}(G)$ by zeroing all entries in the v-th row and u-th column of $\tilde{A}(G)$ and setting the entry (v,u) to 1.

Denote by $\mathcal{P}_{u,v}(G)$ the set of all simple $u \to v$ paths in G. We extend μ to paths and set $\mu(P) = \prod_{i=1}^{\ell-1} -x_{v_i,v_{i+1}}$ if $P = v_1 \dots v_\ell$.

Lemma 6.11. *For any $v, u \in V$, we have:*

$$\det(\tilde{A}^{v,u}(G)) = \sum_{P \in \mathcal{P}_{u,v}} \mu(P) \cdot \det(\tilde{A}(G[V \setminus V(P)])).$$

Proof. TOPROVE 12 □

Since $\tilde{A}(G)$ is invertible, we have

$$\tilde{A}(G)^{-1}_{u,v} = \frac{(-1)^{u+v}}{\det(\tilde{A}(G))} \cdot \det(\tilde{A}^{v,u}(G)),$$

In particular, as noted by Sankowski [San04], since $\det(\tilde{A}(G[Z])) \not\equiv 0$ for any $Z \subseteq V$, it follows by Lemma 6.11 that $\tilde{A}(G)^{-1}_{v,u}$ is a zero polynomial if and only if $\sum_{P \in \mathcal{P}_{u,v}} \mu(P)$ is a zero polynomial, or in other words, if there is no $u \to v$ path in G.

We are now ready to prove Theorem 6.5. Recall that $F = \{u_1v_1, \dots, u_kv_k\}$ is a group of parallel inter-SCC edges, that is, $u_i \in R$, $v_i \in T$, where R and T are distinct SCCs of G. Let $r \in R$ and $t \in T$ be arbitrarily chosen.

Let us first assume that F is non-redundant. Then, for all simple $r \to t$ paths P, P is of the form $P_R \cdot f \cdot P_T$, where $f = u_iv_i \in F$, P_R is a simple (possibly empty) $r \to u_i$ path entirely contained in R, and P_T is a simple $v_i \to t$ path entirely contained in T. Note that neither P can use two edges from F at once, nor it can enter R after leaving it for the first time, nor it can leave T once that SCC is entered. So, by Lemma 6.11, we have

$$\begin{aligned}
\det(\tilde{A}^{t,r}(G)) &= \sum_{P \in \mathcal{P}_{r,t}} \mu(P) \cdot \det(\tilde{A}(G[V \setminus V(P)])) \\
&= \sum_{i=1}^{k} \sum_{P_R \in \mathcal{P}_{r,u_i}} \sum_{P_T \in \mathcal{P}_{v_i,t}} \mu(P_R \cdot u_iv_i \cdot P_T) \cdot \det(\tilde{A}(G[V \setminus V(P_R) \setminus V(P_T)])) \\
&= -\sum_{i=1}^{k} x_{u_i,v_i} \sum_{P_R \in \mathcal{P}_{r,u_i}} \sum_{P_T \in \mathcal{P}_{v_i,t}} \mu(P_R) \cdot \mu(P_T) \cdot \det(\tilde{A}(G[V \setminus V(P_R) \setminus V(P_T)])).
\end{aligned}$$

Let $S_1, \dots, S_s$ be the SCCs of G. Since every path P_R (P_S) that the sum iterates through is contained in the SCC R (T, resp.), and $R \neq T$, using Lemma 6.9 and Corollary 6.10 applied to the graphs G and $G[V \setminus V(P_R) \setminus V(P_T)]$, we obtain:

$$\begin{aligned}\det(\tilde{A}^{t,r}(G)) &= -\sum_{i=1}^{k} x_{u_i,v_i} \sum_{P_R \in \mathcal{P}_{r,u_i}} \sum_{P_T \in \mathcal{P}_{v_i,t}} \mu(P_R) \cdot \mu(P_T) \cdot \left(\prod_{S_i \notin \{R,T\}} \det(\tilde{A}(G[S_i])) \right) \cdot \\ &\qquad \det(\tilde{A}(G[R \setminus V(P_R)])) \cdot \det(\tilde{A}(G[T \setminus V(P_T)])) \\ &= -\det(\tilde{A}(G)) \sum_{i=1}^{k} x_{u_i,v_i} \left(\frac{1}{\det(\tilde{A}(G[R]))} \sum_{P_R \in \mathcal{P}_{r,u_i}} \mu(P_R) \cdot \det(\tilde{A}(G[R \setminus V(P_R)])) \right) \cdot \\ &\qquad \left(\frac{1}{\det(\tilde{A}(G[T]))} \sum_{P_T \in \mathcal{P}_{v_i,t}} \mu(P_T) \cdot \det(\tilde{A}(G[T \setminus V(P_T)])) \right).\end{aligned}$$

Therefore, by the relationship between the inverse and the adjoint:

$$\begin{aligned}\tilde{A}(G)^{-1}_{r,t} = (-1)^{r+t} \sum_{i=1}^{k} -x_{u_i,v_i} &\left(\frac{1}{\det(\tilde{A}(G[R]))} \sum_{P_R \in \mathcal{P}_{r,u_i}} \mu(P_R) \cdot \det(\tilde{A}(G[R \setminus V(P_R)])) \right) \cdot \\ &\left(\frac{1}{\det(\tilde{A}(G[T]))} \sum_{P_T \in \mathcal{P}_{v_i,t}} \mu(P_T) \cdot \det(\tilde{A}(G[T \setminus V(P_T)])) \right).\end{aligned} \tag{1}$$

On the other hand, again by Lemmas 6.9 and 6.11, we have

$$\begin{aligned}\tilde{A}(G)^{-1}_{r,u_i} &= \frac{(-1)^{r+u_i}}{\det(\tilde{A}(G))} \cdot \det(\tilde{A}^{u_i,r}(G)) \\ &= \frac{(-1)^{r+u_i}}{\det(\tilde{A}(G))} \cdot \sum_{P \in \mathcal{P}_{r,u_i}} \mu(P) \cdot \det(\tilde{A}(G[V \setminus V(P)])) \\ &= \frac{(-1)^{r+u_i}}{\det(\tilde{A}(G))} \cdot \sum_{P \in \mathcal{P}_{r,u_i}} \mu(P) \cdot \det(\tilde{A}(G[R \setminus V(P)])) \cdot \prod_{S_i \neq R} \det(\tilde{A}(G[S_i])) \\ &= \frac{(-1)^{r+u_i}}{\det(\tilde{A}(G))} \cdot \frac{\det(\tilde{A}(G))}{\det(\tilde{A}(G[R]))} \sum_{P \in \mathcal{P}_{r,u_i}} \mu(P) \cdot \det(\tilde{A}(G[R \setminus V(P)])).\end{aligned}$$

and thus

$$(-1)^{r+u_i} \cdot \tilde{A}(G)^{-1}_{r,u_i} = \frac{1}{\det(\tilde{A}(G[R]))} \sum_{P \in \mathcal{P}_{r,u_i}} \mu(P) \cdot \det(\tilde{A}(G[R \setminus V(P)])). \tag{2}$$

Similarly, we can obtain

$$(-1)^{t+v_i} \cdot \tilde{A}(G)^{-1}_{v_i,t} = \frac{1}{\det(\tilde{A}(G[T]))} \sum_{P \in \mathcal{P}_{v_i,t}} \mu(P) \cdot \det(\tilde{A}(G[T \setminus V(P)])). \tag{3}$$

By plugging in Equations (2) and (3) into Equation (1), we obtain:

$$\begin{aligned}\tilde{A}(G)^{-1}_{r,t} &= (-1)^{r+t} \sum_{i=1}^{k} -x_{u_i,v_i} \cdot (-1)^{r+u_i} \cdot \tilde{A}(G)^{-1}_{r,u_i} \cdot (-1)^{t+v_i} \cdot \tilde{A}(G)^{-1}_{v_i,t} \\ &= -\sum_{i=1}^{k} (-1)^{u_i+v_i} \cdot x_{u_i,v_i} \cdot \tilde{A}(G)^{-1}_{r,u_i} \cdot \tilde{A}(G)^{-1}_{v_i,t},\end{aligned}$$

which proves the “ $\Longleftarrow$ ” implication of Theorem 6.5.

Now suppose the edges F are redundant. It is enough to prove

$$\det(\tilde{A}(G)) \cdot \tilde{A}(G)^{-1}_{r,t} \not\equiv -\det(\tilde{A}(G)) \sum_{i=1}^{k} (-1)^{u_i+v_i} \cdot x_{u_i,v_i} \cdot \tilde{A}(G)^{-1}_{r,u_i} \cdot \tilde{A}(G)^{-1}_{v_i,t},$$

or, by multiplying both sides by $\det(\tilde{A}(G))$, equivalently

$$\det(\tilde{A}(G)) \cdot \det(\tilde{A}^{t,r}(G)) \not\equiv -\sum_{i=1}^{k} x_{u_i,v_i} \cdot \det(\tilde{A}^{u_i,r}(G)) \cdot \det(\tilde{A}^{t,v_i}(G)), \tag{4}$$

We prove that the polynomial on the left-hand side of Equation (4) contains a monomial that the right-hand side polynomial lacks. Namely, let P be some simple $r \to t$ path that goes through an SCC Y of G such that $Y \neq R$ and $Y \neq T$. Note that P does not go through any of edges in F, since all vertices of $P \cap Y$ have to appear on P after all vertices of $P \cap R$, and before all vertices of $P \cap T$. By Lemma 6.11, the left-hand side contains a monomial

$$\left(\prod_{i=1}^{n} x_{i,i}\right) \cdot \mu(P) \cdot \left(\prod_{i \in V \setminus V(P)} x_{i,i}\right).$$

However, each monomial in the right-hand side polynomial has a variable of the form x_{u_i,v_i}, where $u_iv_i \in F$, whereas the above monomial clearly does not contain such variables by $E(P) \cap F = \emptyset$.

A Decremental Single Source Reachability on DAGs

In this section, we explain a decremental data structure that maintains single source reachability information on DAG $G = (V, E)$, as summarized in the following lemma. The data structure extends that of Italiano [Ita88] and is equipped with the operations required in Section 5.1.

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:

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

The Data Structure. For every vertex $y \neq r$, we define a doubly linked list active$[y]$ consisting of incoming edges of y in G. The data structure maintains $p(y)$, which points to the first edge in active$[y]$ that connects y to Desc^r, and $c(y)$, which points to the second edge in active$[y]$ that connects y to Desc^r. If no such edge exists in active$[y]$, we set the respective pointer to be null.

We now introduce the two invariants of the data structure.

Invariant A.1. *For every vertex $y \neq r$, if $p(y) =$ null, then y is not reachable from r.*

Invariant A.2. *For every vertex $y \neq r$, if $c(y) =$ null, then r can reach y through at most one edge.*

Initialization. To compute Desc^r, we simply compute a reachability tree rooted at r. For every vertex $y \neq r$, we set $\mathrm{active}[y]$ to be the list of all incoming edges of y in G.

After moving a vertex in $\mathrm{active}[y] \cap \mathrm{Desc}^r$ to the front of $\mathrm{active}[y]$ (if such a vertex exists), we set $p(y)$ to point to the front of $\mathrm{active}[y]$. To initialize $c(y)$, we first point $c(y)$ to the front element of $\mathrm{active}[y]$, and then call $\textsc{UpdateC}(y)$ in Algorithm 1 to find the first edge in $\mathrm{active}[y]$ satisfying the definition of $c(y)$.

Set D^r would maintain the set of vertices removed from Desc^r due to the last update. We initialize $\mathrm{D}^r = \emptyset$.

Handling Edge Deletions. Assume that a deletion of edges E_{del} has happened as the last update. To maintain Desc^r, we use the queue Q, containing the edges which their tail needs to be reconnected to Desc^r after the removal of the edge. We begin by setting $Q = E_{\mathrm{del}}$.

While $Q \neq \emptyset$, for edge $xy \in Q$, the algorithm checks whether the removal of xy affects Invariants A.1 and A.2. The maintenance procedure is as follows.

1. If $xy = p(y)$, then y loses its connection to Desc^r. In this case, we update $p(y)$ as follows.
   - If $c(y) \neq \mathsf{null}$, we remove the elements in the front of $\mathrm{active}[y]$ until we get $p(y) = c(y)$. By the definition of pointers, this ensures that Invariant A.1 is correctly maintained. To correctly maintain $c(y)$, we update $c(y)$ by calling $\textsc{UpdateC}(y)$, which finds the first edge in $\mathrm{active}[y]$, after $p(y)$, that can connect y to Desc^r, as desired.
   - If $c(y) = \mathsf{null}$, then xy was the only edge connecting y to Desc^r. Thus, we set $p(y) \leftarrow \mathsf{null}$, remove y from Desc^r and add it to D^r. Since the children of y may be connected to Desc^r through y, we add all outgoing edges of y to Q.
2. If $xy = c(y)$, we update $c(y)$ by calling $\textsc{UpdateC}(y)$ to correctly maintain $c(y)$, and then remove xy from $\mathrm{active}[y]$ to correctly maintain $\mathrm{active}[y]$ as a subset of incoming edges of y.
3. If $xy \neq p(y)$ and $xy \neq c(y)$, we only need to remove xy from $\mathrm{active}[y]$.

We conclude this section by the following lemma.

Lemma A.1. *Given an m-edge DAG $G = (V, E)$, there is a decremental data structure that maintains the set Desc^r of vertices reachable from the root vertex r in $O(m)$ total update time, and supports the following additional operation:*

– $\textsc{In}(y)$: *Return* `True` *if $y \neq r$ and y has an in-neighbor from $\mathrm{Desc}^r \setminus r$, and* `False` *otherwise.*

Additionally, it maintains the set D^r of vertices that have been removed from Desc^r due to the most recent deletion.

Proof. TOPROVE 13 □

Extending the Data Structure. We extend the data structure of Lemma A.1 to support all the operations of Lemma 5.1. We define Anc^r to be the set of vertices that can reach r, and A^r as the set of vertices that are removed from Anc^r due to the last deletion of edges E_{del}.

Proof. TOPROVE 14 □

B Decremental Single Source Reachability on General Graphs

In this section, we explain a data structure that maintains a decremental single source reachability tree T rooted at r on a graph $G = (V, E)$. The guarantees are stated in the following lemma.

```
Algorithm 1: DECREMENTAL-SINGLE-SOURCE-REACHABILITY-ON-DAGS
  Input: a DAG $G = (V, E)$ and a root vetex $r \in V$
  Maintain: set $\mathrm{Desc}^r \subseteq V$ of vertices reachable from $r$ and set $\mathrm{D}^r$ of vertices removed from
            $\mathrm{Desc}^r$ due to the last deletion
1  Procedure INITIALIZE
2      $\mathrm{Desc}^r \leftarrow$ vertices reachable from $r$
3      foreach vertex $y \neq r$ do
4          active$[y] \leftarrow$ incoming edges of $y$ in $E$
5          move a vertex (if any) in active$[y] \cap \mathrm{Desc}^r$ to the front of active$[y]$
6          set $p(y)$ and $c(y)$ to point to the front of active$[y]$
7          UPDATEC$(y)$.
8  Procedure DELETE$(E_{\mathrm{del}})$
9      $E \leftarrow E \setminus E_{\mathrm{del}}$
10     $Q \leftarrow E_{\mathrm{del}}$
11     $\mathrm{D}^r \leftarrow \emptyset$
12     while $Q \neq \emptyset$ do
13         $xy :=$ DEQUEUE$(Q)$
14         if $xy = p(y)$ then
15             UPDATEP$(y)$
16         else if $xy = c(x)$ then
17             UPDATEC$(y)$
18         else
19             remove $xy$ from active$[y]$
20 Procedure UPDATEP$(y)$
21     if $c(y) =$ null then
22         $p(y) \leftarrow$ null
23         add $y$ to $\mathrm{D}^r$
24         foreach outgoing edge $yz \in E$ do
25             ENQUEUE$(Q, yz)$
26     else
27         remove items from the front of active$[y]$ until $p(y) = c(y)$
28         UPDATEC$(y)$
29 Procedure UPDATEC$(y)$
30     move pointer $c(y)$ to the next element in active$[y]$
31     if $c(y) = zy$ such that $z \neq r$ and $p(z) =$ null then
32         UPDATEC$(y)$
33 Procedure IN$(y)$
34     if $p(y)$ or $c(y)$ points to an edge different than $ry$ then
35         return True
36     else
37         return False
```

Lemma 5.5. *Given a directed graph $G = (V, E)$ and the subgraphs G^r for every $r \in V$ 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 that maintains the SCCs and the set of parallel inter-SCC edges of all graphs in $O(m + n \log n)$ amortized update time, where m is the number of edges in the current graph G.*

Additionally, for all $r \in V$, the data structure maintains the sets D^r *and* A^r *of vertices removed from* Desc^r *and* Anc^r*, respectively, as a consequence of the last update and supports the following operations in time $O(1)$, where X, Y, R are the SCCs containing x, y, r, respectively:*

- $\textsc{In}(y, r)$: *return* `True` *if, in G^r, $Y \neq R$ and Y has an in-neighbor from* $\mathrm{Desc}^r \setminus R$, *and* `False` *otherwise.*
- $\textsc{Out}(x, r)$: *return* `True` *if, in G^r, $X \neq R$ and X has an out-neighbor to* $\mathrm{Desc}^r \setminus R$, *and* `False` *otherwise.*

Lastly, the data structure maintains the set S^r *of the SCCs in G^r such that* $\textsc{In}(y, r)$ *or* $\textsc{Out}(x, r)$ *has changed from* `True` *to* `False` *due to the last deletion.*

Our algorithm builds upon that of Roditty and Zwick, which we briefly explain in Appendix B.1 before proceeding with our extension in Appendix B.2.

B.1 The data structure of [RZ16]

The data structure uses *uninspected* inter-SCC edges to maintain T. An uninspected edge is either a tree edge, which will remain uninspected, or it is not useful; once it has been inspected by the data structure, it will no longer be uninspected. Initially, all edges are uninspected.

A vertex z is called *active* if $\mathrm{active}[Z] \neq \mathsf{null}$. For a vertex $z \in V$, $\mathrm{in}[z]$ is the set of uninspected inter-SCC edges entering z, and $\mathrm{out}[z]$ is the set of *all* edges outgoing z. For every SCC Z, the data structure maintains a doubly linked list $\mathrm{active}[Z]$ that contains all active vertices of Z.

The data structure maintains the following invariant.

Invariant B.1. *If $y \in V \setminus R$ is the first vertex in* $\mathrm{active}[Y]$ *and xy is the first edge in* $\mathrm{in}[y]$*, then xy is the tree edge connecting the SCC Y to T. In particular, if* $\mathrm{active}[Y] = \mathsf{null}$*, then none of the vertices of Y are connected to T.*

The data structure maintains a sequence of graphs $G_0, G_1, \ldots, G_t$, where t is the number of insert operations. Here, G_i is the snapshot of G after the ith insertion. However, edges that are subsequently deleted from G are also deleted from G_i, ensuring that at each step, $\emptyset = E_0 \subseteq E_1 \subseteq \cdots \subseteq E_t = E$, where E_i is the set of edges in G_i. The data structure maintains an array indexed by V to maintain the SCCs: for every vertex $z \in V$, $\mathrm{scc}_i[z]$ is the name of the SCC containing z in G_i. The guarantees of the data structure are summarized in the following theorem.

Theorem B.1 (Sections 3 and 4 of [RZ16], rephrased)**.** *Given a directed graph $G = (V, E)$ and the sequence of subgraphs $G_0, G_1, \ldots, G_t$ defined above, there is a fully dynamic data structure that maintains the SCCs for each G_i, and supports each insert operation on G in $O(m + n \log n)$ worst-case time and each delete operation on G in $O(m + n \log n)$ amortized update time, where m is the number of edges in the current graph G.*

Moreover, the data structure supports the following additional operations:

- $\textsc{Detect}()$: *List all the components that decomposed as a consequence of the most recent delete operation in G, together with the index i specifying the subgraph G_i that the decomposition happened. This operation runs in $O(n)$ time.*

– List(Z, i): *Given an SCC Z in G_i, list all the SCCs Z decomposed as a consequence of the most recent delete operation. This operation runs in time proportional to the number of the SCCs that Z decomposed into.*

Handling Edge Insertions. After an insertion, we can afford to recompute the SCCs in the current graph, i.e., in G_t, as well as all the values required by the data structure, such as $\mathrm{in}[\cdot]$ and $\mathrm{out}[\cdot]$. It is easy to see that this takes $O(m + n \log n)$ time.

Handling Edge Deletions. We explain how to efficiently maintain the SCCs of G_i after the deletion of E_{del}. Using the operation Detect() from Theorem B.1, we obtain the SCCs of G_i that has decomposed after the deletion. Assume that SCC Z is one of them, and List(Z, i) has returned $Z_1, Z_2, \ldots, Z_k$ as the SCCs Z decomposed into, where $|Z_1| \geq |Z_2| \geq \cdots \geq |Z_k|$. We first show how to maintain $\mathrm{active}[z_j]$. Instead of scanning $\mathrm{active}[Z]$ and moving each vertex to a new list, we let Z_1 to inherit $\mathrm{active}[Z]$. Thus, we only need to move the vertices that does not belong to Z_1, resulting in $O(\sum_{j=2}^{k} |Z_j|)$ time for this operation. Note that we can simultaneously update $\mathrm{in}[z]$ for every vertex $z \in Z$. Since $|Z_1| \geq |Z_j|$, during a sequence of edge deletions, each vertex can be moved at most $\log n$ times, which results in the following lemma.

Lemma B.2 (Lemma 5.1 of [RZ16])**.** *The total cost to maintain* $\mathrm{active}[\cdot]$ *in each graph G_i during any sequence of edge deletions is $O(m+n \log n)$, where m is the number of edges in the initial graph G_i.*

To maintain $\mathrm{in}[\cdot]$, we add each SCC that may need to be inspected to Q. If SCC Z has decomposed into $Z_1, Z_2, \ldots, Z_k$, we then add all Z_j to Q, except for the (possibly) one that contains the root r. Also, for every edge $xy \in E_{\mathrm{del}}$, if y is the first vertex in $\mathrm{active}[Y]$ and xy is the first edge in $\mathrm{in}[y]$, we then add the SCC Y to Q. We then pick an SCC $W \in Q$. Let w be the first vertex in $\mathrm{active}[W]$. We scan $\mathrm{in}[w]$ to find an inter-SCC incoming edge xw that satisfies the following conditions: it should be removed from G_i, and either $\mathrm{active}[\mathrm{scc}_i[x]] \neq \emptyset$ or $\mathrm{scc}_i[x] = \mathrm{scc}_i[r]$. We keep removing edges from $\mathrm{in}[w]$ until we find such an edge. If we reach $\mathrm{in}[w] = \mathsf{null}$, we remove w from $\mathrm{active}[W]$, choose the first vertex in $\mathrm{active}[W]$ again, and try to find an edge that connects W to T. If we reach $\mathrm{active}[W] = \mathsf{null}$ without finding a connecting edge, then W is no longer connected to T. In this case, for every vertex $w \in W$ and every outgoing edge $wy \in \mathrm{out}[w]$, if y is the first vertex in $\mathrm{active}[Y]$ and wy is the first edge in $\mathrm{in}[y]$, we then add the SCC Y to Q. We conclude this subsection with the following lemma which guarantees the correctness of the algorithm.

Lemma B.3 (Lemma 5.2 of [RZ16])**.** *The algorithm described above correctly maintains T in G_i (and so* $\mathrm{in}[\cdot]$ *and* $\mathrm{active}[\cdot]$*) during any sequence of edge deletions in $O(m)$ total update time, where m is the number of edges in the initial graph G_i.*

B.2 Our extension to the data structure

Let G^u be the snapshot of G taken after the last insertion E_u centered around u (if such an insertion occurred). Note that subsequent edge insertions centered around vertices *different* from u do not alter G^u. However, edges that are subsequently deleted from G are also deleted from G^u, ensuring that at each step, $E^u \subseteq E$. Thus, G^u undergoes only edge deletions. If vertex u serves as an insertion center in future updates, the snapshot graph G^u is reinitialized, and T is constructed from scratch.

Note that G^u is actually the graph G_i in the sequence defined in Appendix B.1, where i is the *last* time that an insertion happened around u. We can easily maintain G^u's by maintaining the sequence $G_0, G_1, \ldots, G_t$ and bookkeeping the time index i in the sequence for each vertex u. After an insertion around u, we update the index to the most recent t.

Similar to Section 5.2, we define Desc^u as the set of vertices reachable from u, and Anc^u as the set of vertices that can reach u in G^u. We extend the data structure of Appendix B.1 to support the following additional operations.

– $\textsc{In}(y, r)$: for any vertex y with $Y \neq R$, return `True` if Y has an in-neighbor from $\mathrm{Desc}^r \setminus R$ In G^r, and `False` otherwise. Here, R, Y are the SCCs that contain r, y in G^r, respectively.

– $\textsc{Out}(x, r)$: for any vertex x with $X \neq R$, return `True` if R has an out-neighbor to $\mathrm{Anc}^r \setminus X$ In G^r, and `False` otherwise. Here, R, X are the SCCs that contain r, x in G^r, respectively.

Here, we maintain T in each G^u. To adapt this to the data structure of Appendix B.1, for each G_i, if it is a snapshot of an insertion around vertex u, we then simply initialize T, and maintain the related values decrementally. Note that, by Lemmas B.2 and B.3, we can freely choose the root vertex r, which in this case is $r = u$.

To implement the operations, we take advantage of the same idea we used in Appendix A, but now on the SCCs instead of the vertices: for each SCC Y, we maintain the two pointers $p(Y)$, which points to an edge x_2y such that $x_2 \in V(T)$ and $\mathrm{scc}_r[x_2] \neq \mathrm{scc}_r[x_1]$, and $c(Y)$, which points to an edge x_2y such that $x_2 \in V(T)$ and $\mathrm{scc}_r[x_2] \neq \mathrm{scc}_r[x_1]$. If no such edge exists in $p(Y)$ or $c(Y)$, we set the respective pointer to be null. Roughly speaking, $p(Y)$ and $c(Y)$ maintain two *different* (if any) incoming edges x_1y and x_2y of Y such that x_1 and x_2 belong to $V(T)$ from different SCCs.

The data structure maintains $\mathrm{active}[Y]$ such that $p(Y)$ always points to the front of $\mathrm{in}[y]$, where y is the first vertex of $\mathrm{active}[Y]$. Also, $c(Y)$ always points to the first edge *after* $\mathrm{in}[y]$ that satisfies the criteria defining $c(Y)$. We now introduce the other invariant of our data structure.

Invariant B.2. *If $y \in V \setminus R$, then $c(Y)$ is an edge coming from an SCC different than $p(Y)$ and connects Y to T (if any). If $c(Y) = \mathsf{null}$, then Y has at most one SCC connecting it to T.*

Similar to Appendix A, handling the sets Anc^r and A^r, and the operation $\textsc{Out}(x, r)$ is simply done on the reverse graph of G^u. Thus, here, we only discuss how to handle the sets Desc^r and D^r, and the operation $\textsc{In}(y, r)$ as follows.

– Handling $\mathrm{Desc}^u = V(T)$ is simply done by the data structure. If an SCC Y got removed during the update, i.e., we reach $\mathrm{active}[Y] = \mathsf{null}$, we then add all the vertices of Y to D^u.

– The operation $\textsc{InNeighbor}(y, r)$ is supported by the pointer $c(Y)$. By the definition of the pointer, if $c(Y) = \mathsf{null}$, we return `False` in response to the operation, and `True` otherwise.

Handling the insertions and deletions are similar to the data structure of Appendix A.

References

[ADDJS93] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. *On Sparse Spanners of Weighted Graphs*. *Discret. Comput. Geom.* 9, pages 81–100, 1993 (cited on pages 1, 4).

[ADK13] 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 (cited on page 2).

[ADKKP16] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. *On Fully Dynamic Graph Sparsifiers*. In: *57th FOCS*, pages 335–344, 2016 (cited on page 4).

[AGU72] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. *The Transitive Reduction of a Directed Graph*. *SIAM J. Comput.* 1.2, pages 131–137, 1972 (cited on pages 1, 4, 5).

[AKLPRT23] Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. *Optimal Decremental Connectivity in Non-Sparse Graphs*. In: *50th ICALP*. Vol. 261, 6:1–6:17, 2023 (cited on page 4).

[BBGNSSS22] 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 ICALP*. Vol. 229, 20:1–20:20, 2022 (cited on page 4).

[BFH21] 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 (cited on page 4).

[BGW19] 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, pages 128–155, 2019 (cited on page 7).

[BK15] András A. Benczúr and David R. Karger. *Randomized Approximation Schemes for Cuts and Flows in Capacitated Graphs*. *SIAM J. Comput.* 44.2, pages 290–319, 2015 (cited on page 4).

[BK96] András A. Benczúr and David R. Karger. *Approximating* s-t *Minimum Cuts in* $\tilde{O}(n^2)$ *Time*. In: *28th STOC*. ACM, pages 47–55, 1996 (cited on pages 1, 4).

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. *Fully dynamic randomized algorithms for graph spanners*. *ACM Trans. Algorithms* 8.4, 35:1–35:51, 2012 (cited on page 4).

[BNS19] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. *Dynamic Matrix Inverse: Improved Algorithms and Matching Conditional Lower Bounds*. In: *60th FOCS*, pages 456–480, 2019 (cited on pages 5, 6, 8).

[BOWLH12] 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, page 281, 2012 (cited on page 2).

[Bra] Jan van den Brand. *Complexity Term Balancer*. `www.ocf.berkeley.edu/~vdbrand/complexity/`. Tool to balance complexity terms depending on fast matrix multiplication. (cited on page 20).

[BS07] Surender Baswana and Sandeep Sen. *A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs*. *Random Struct. Algorithms* 30.4, pages 532–563, 2007 (cited on page 4).

[CCPS21] Ruoxu Cen, Yu Cheng, Debmalya Panigrahi, and Kevin Sun. *Sparsification of Directed Graphs via Cut Balance*. In: *48th ICALP*. Vol. 198, 45:1–45:21, 2021 (cited on page 4).

[CGLE15] James R. Clough, Jamie Gollings, Tamar V. Loach, and Tim S. Evans. *Transitive reduction of citation networks*. *J. Complex Networks* 3.2, pages 189–203, 2015 (cited on page 2).

[Cha08] Timothy M. Chan. *All-Pairs Shortest Paths with Real Weights in* $O(n^3/\log n)$ *Time*. *Algorithmica* 50.2, pages 236–243, 2008 (cited on page 1).

[CKPPSV16] 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 FOCS*, pages 583–592, 2016 (cited on page 4).

[CT00] Joseph Cheriyan and Ramakrishna Thurimella. *Approximating Minimum-Size k-Connected Spanning Subgraphs via Matching*. *SIAM J. Comput.* 30.2, pages 528–560, 2000 (cited on pages 1, 4).

[CW82] Don Coppersmith and Shmuel Winograd. *On the Asymptotic Complexity of Matrix Multiplication*. *SIAM J. Comput.* 11.3, pages 472–492, 1982 (cited on page 1).

[DB05] 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 (cited on page 2).

[DI05] 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, pages 147–156, 2005 (cited on page 17).

[FG19] Sebastian Forster and Gramoz Goranci. *Dynamic low-stretch trees via dynamic low-diameter decompositions*. In: *51st STOC*, pages 377–388, 2019 (cited on page 4).

[FHHP19] Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. *A General Framework for Graph Sparsification*. *SIAM J. Comput.* 48.4, pages 1196–1223, 2019 (cited on page 4).

[GIKPP16] Loukas Georgiadis, Giuseppe F. Italiano, Aikaterini Karanasiou, Charis Papadopoulos, and Nikos Parotsidis. *Sparse Subgraphs for 2-Connectivity in Directed Graphs*. In: *15th ESA*. Vol. 9685, pages 150–166, 2016 (cited on page 4).

[GKRST91] Phillip B. Gibbons, Richard M. Karp, Vijaya Ramachandran, Danny Soroker, and Robert Endre Tarjan. *Transitive Compaction in Parallel via Branchings*. *J. Algorithms* 12.1, pages 110–125, 1991 (cited on pages 1, 7, 14).

[GSEOYB21] 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 (cited on page 2).

[HHLLX15] 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. Security, Privacy and trust in network-based Big Data, pages 144–156, 2015 (cited on page 2).

[HKRT95] Xiaofeng Han, Pierre Kelsen, Vijaya Ramachandran, and Robert Endre Tarjan. *Computing Minimal Spanning Subgraphs in Linear Time*. *SIAM J. Comput.* 24.6, pages 1332–1358, 1995 (cited on pages 1, 4).

[HLT01] 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, pages 723–760, 2001 (cited on page 4).

[Ita88] Giuseppe F. Italiano. *Finding paths and deleting edges in directed acyclic graphs*. *Information Processing Letters* 28.1, pages 5–11, 1988 (cited on pages 6, 7, 10, 23).

[JRDY12] 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 (cited on page 2).

[KFS10] Steffen Klamt, Robert J. Flassig, and Kai Sundmacher. *TRANSWESD: inferring cellular networks with transitive reduction*. *Bioinform.* 26.17, pages 2160–2168, 2010 (cited on page 2).

[Kin99] Valerie King. *Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs*. In: *40th FOCS*, pages 81–91, 1999 (cited on page 4).

[KS02] Valerie King and Garry Sagert. *A Fully Dynamic Algorithm for Maintaining the Transitive Closure*. *J. Comput. Syst. Sci.* 65.1, pages 150–167, 2002 (cited on page 17).

[LGS12] Bundit Laekhanukit, Shayan Oveis Gharan, and Mohit Singh. *A Rounding by Sampling Approach to the Minimum Size k-Arc Connected Subgraph Problem*. In: *39th ICALP*. Vol. 7391, pages 606–616, 2012 (cited on page 4).

[Men23] 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, pages 7727–7738, 2023 (cited on page 2).

[Net93] Robert H. B. Netzer. *Optimal tracing and replay for debugging shared-memory parallel programs*. *SIGPLAN Not.* 28.12, pages 1–11, 1993 (cited on page 2).

[NI92] 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, pages 583–596, 1992 (cited on page 4).

[PHFFK13] 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, page 73, 2013 (cited on page 2).

[PL87] Johannes A. La Poutré and Jan van Leeuwen. *Maintenance of Transitive Closures and Transitive Reductions of Graphs*. In: *13th WG*. Vol. 314, pages 106–120, 1987 (cited on pages 2, 3).

[PS89] David Peleg and Alejandro A. Schäffer. *Graph spanners*. *J. Graph Theory* 13.1, pages 99–116, 1989 (cited on page 4).

[Rod08] Liam Roditty. *A faster and simpler fully dynamic transitive closure*. *ACM Trans. Algorithms* 4.1, 6:1–6:16, 2008 (cited on page 5).

[RTZ08] 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 (cited on page 4).

[RZ08] Liam Roditty and Uri Zwick. *Improved Dynamic Reachability Algorithms for Directed Graphs*. *SIAM J. Comput.* 37.5, pages 1455–1471, 2008 (cited on pages 6, 7).

[RZ16] 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, pages 712–733, 2016 (cited on pages 5, 6, 7, 8, 12, 13, 26, 27).

[San04] Piotr Sankowski. *Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract)*. In: *45th FOCS*, pages 509–517, 2004 (cited on pages 5, 6, 8, 16, 17, 18, 20, 21).

[San05] Piotr Sankowski. *Subquadratic Algorithm for Dynamic Shortest Distances*. In: *11th International Computing and Combinatorics Conference (COCOON)*. Vol. 3595, pages 461–470, 2005 (cited on pages 6, 16).

[SS11] Daniel A. Spielman and Nikhil Srivastava. *Graph Sparsification by Effective Resistances*. *SIAM J. Comput.* 40.6, pages 1913–1926, 2011 (cited on pages 1, 4).

[ST11] Daniel A. Spielman and Shang-Hua Teng. *Spectral Sparsification of Graphs*. *SIAM J. Comput.* 40.4, pages 981–1025, 2011 (cited on pages 1, 4).

[Str69] Volker Strassen. *Gaussian elimination is not optimal*. *Numerische Mathematik* 13.4, pages 354–356, 1969 (cited on page 1).

[STZ24] Sushant Sachdeva, Anvith Thudi, and Yibin Zhao. *Better Sparsifiers for Directed Eulerian Graphs*. In: *51st ICALP*. Vol. 297, 119:1–119:20, 2024 (cited on page 4).

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. *High-Probability Parallel Transitive-Closure Algorithms*. *SIAM J. Comput.* 20.1, pages 100–125, 1991 (cited on pages 8, 19).

[Vas12] Virginia Vassilevska Williams. *Multiplying matrices faster than coppersmith-winograd*. In: *44th STOC*, pages 887–898, 2012 (cited on page 1).

[Vet01] Adrian Vetta. *Approximating the minimum strongly connected subgraph via a matching lower bound*. In: *12th SODA*, pages 417–426, 2001 (cited on page 4).

[VXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. *New Bounds for Matrix Multiplication: from Alpha to Omega*. In: *35th SODA*, pages 3792–3835, 2024 (cited on pages 1, 6).

[XHB06] 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 (cited on page 2).