Decremental Data Structures for Connectivity and Dominators in Directed Graphs*†

Loukas Georgiadis[1], Thomas Dueholm Hansen[2], Giuseppe F. Italiano[3], Sebastian Krinninger[4], and Nikos Parotsidis[5]

1 **University of Ioannina, Ioannina, Greece**
`loukas@cs.uoi.gr`
2 **Aarhus University, Aarhus, Denmark**
`tdh@cs.au.dk`
3 **University of Rome Tor Vergata, Rome, Italy**
`giuseppe.italiano@uniroma2.it`
4 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`sebastian.krinninger@univie.ac.at`
5 **University of Rome Tor Vergata, Rome, Italy**
`nikos.parotsidis@uniroma2.it`

Abstract

We introduce a new dynamic data structure for maintaining the strongly connected components (SCCs) of a directed graph (digraph) under edge deletions, so as to answer a rich repertoire of connectivity queries. Our main technical contribution is a decremental data structure that supports sensitivity queries of the form "are u and v strongly connected in the graph $G \setminus w$?", for any triple of vertices u, v, w, while G undergoes deletions of edges. Our data structure processes a sequence of edge deletions in a digraph with n vertices in $O(mn \log n)$ total time and $O(n^2 \log n)$ space, where m is the number of edges before any deletion, and answers the above queries in constant time. We can leverage our data structure to obtain decremental data structures for many more types of queries within the same time and space complexity. For instance for edge-related queries, such as testing whether two query vertices u and v are strongly connected in $G \setminus e$, for some query edge e.

As another important application of our decremental data structure, we provide the first nontrivial algorithm for maintaining the dominator tree of a flow graph under edge deletions. We present an algorithm that processes a sequence of edge deletions in a flow graph in $O(mn \log n)$ total time and $O(n^2 \log n)$ space. For reducible flow graphs we provide an $O(mn)$-time and $O(m + n)$-space algorithm. We give a conditional lower bound that provides evidence that these running times may be tight up to subpolynomial factors.

1998 ACM Subject Classification E.1 [Graphs and Networks] Trees, F.2.2 Computations on Discrete Structures, G.2.2 [Graph Algorithms] Trees

Keywords and phrases dynamic graph algorithms, decremental algorithms, dominator tree, strong connectivity under failures

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.42

* Full version of this paper available at `https://arxiv.org/abs/1704.08235`.

† The work of L. Georgiadis and T. D. Hansen was partially done while visiting University of Rome Tor Vergata. T. D. Hansen was supported by the Carlsberg Foundation, grant no. CF14-0617. G. F. Italiano was partially supported by the Italian Ministry of Education, University and Research, under Project AMANDA. The work of S. Krinninger was partially done while visiting University of Rome Tor Vergata and while at Max Planck Institute for Informatics, Saarland Informatics Campus, Germany.

44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl;
Article No. 42; pp. 42:1–42:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Dynamic graph algorithms have been extensively studied for several decades, and many important results have been achieved for dynamic versions of fundamental problems, including connectivity, 2-edge and 2-vertex connectivity, minimum spanning tree, transitive closure, and shortest paths (see, e.g., the survey in [14]). We recall that a dynamic graph problem is said to be *fully dynamic* if it involves both insertions and deletions of edges, *incremental* if it only involves edge insertions, and *decremental* if it only involves edge deletions.

The decremental strongly connected components (SCCs) problem asks us to maintain, under edge deletions in a directed graph G, a data structure that given two vertices u and v answers whether u and v are strongly connected in G. We extend this problem to sensitvity queries of the form "are u and v strongly connected in the graph $G \setminus w$?", for any triple of vertices u, v, w, i.e., we additionally allow the query to temporarily remove a third vertex w. We show that this extended decremental SCC problem can be used to answer fast a rich repertoire of connectivity queries, and we present a new and efficient data structure for the problem. In particular, our data structure for the extended decremental SCC problem can be used to support edge-related queries, such as maintaining the strong bridges of a digraph, testing whether two query vertices u and v are strongly connected in $G \setminus e$, reporting the SCCs of $G \setminus e$, or the largest and smallest SCCs in $G \setminus e$, for any query edge e. Furthermore, using our framework, it is possible to maintain the 2-vertex-and 2-edge-connected components of a digraph under edge deletions. All of these extensions can be handled with the same time and space bounds as for the extended decremental SCC problem. (Most of these reductions have been deferred to the full version of the paper.)

A naive approach to solving the extended decremental SCC problem is to maintain separately the SCCs in every subgraph $G \setminus w$ of G, for all vertices w. After an edge deletion we then update the SCCs of all these n subgraphs, where n is the number of vertices in G. If we simply perform a static recompution after each deletion, then we, for example, obtain decremental algorithms with $O(m^2 n)$ total time and $O(n^2)$ space by recomputing the SCCs in each $G \setminus w$ [37] or $O(m^2 + mn)$ total time and $O(m + n)$ space by constructing a more suitable static connectivity data structure [22], respectively. Here m denotes the initial number of edges. The current fastest (randomized) decremental SCC algorithm by Chechik et al. [10] trivially gives $O(mn^{3/2} \log n)$ total update time and $O(mn)$ space for our extended decremental SCC problem.

The main technical contribution of this paper is a data structure for the extended decremental SCC problem with $O(mn \log n)$ total update time that uses $O(n^2 \log n)$ space, and that answers queries in constant time. We obtain this data structure by extending Łącki's decremental SCC algorithm [30]. His algorithm maintains the SCCs of a graph under edge deletions by recursively decomposing the SCCs into smaller and smaller subgraphs. We therefore refer to his data structure as an SCC-decomposition. His total update time is $O(mn)$ and the space used is $O(m + n)$. We observe that the naive algorithm based on SCC-decompositions can be implemented in such a way that most of the work performed is redundant. We obtain our data structure by merging n SCC-decompositions into one joint data structure, which we refer to as a joint SCC-decomposition. Our data structure, like that of Łącki, is deterministic. Using completely different techniques, Georgiadis et al. [21] showed how to answer the same sensitivity queries in $O(mn)$ total time in the incremental setting, i.e., when the input digraph undergoes edge insertions only.

The extended SCC problem is related to the so-called *fault-tolerant model.* Here, one wishes to preprocess a graph G into a data structure that is able to answer fast certain sensitivity queries, i.e., given a failed vertex w (resp., failed edge e), compute a specific property of the subgraph $G \setminus w$ (resp., $G \setminus e$) of G. Our data structure supports sensitivity queries when a digraph G undergoes edge deletions, which gives an aspect of *decremental fault-tolerance*. This may be useful in scenarios where we wish to find the best edge whose deletion optimizes certain properties (fault-tolerant aspect) and then actually perform this deletion (decremental aspect). This is, e.g., done in the computational biology applications considered by Mihalák et al. [32]. Their recursive deletion-contraction algorithm repeatedly finds the edge of a strongly connected digraph whose deletion maximizes quantities such as the number of resulting SCCs or minimizes their maximum size.

As another important application of our joint SCCs data structure, we provide the first nontrivial algorithm for maintaining the dominator tree of a flow graph under edge deletions. A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex $s \in V$, w.l.o.g. containing only vertices reachable from s. A vertex w *dominates* a vertex v (w is a *dominator* of v) if every path from s to v includes w. The *immediate dominator* of a vertex v, denoted by $d(v)$, is the unique vertex that dominates v and is dominated by all dominators of v. The *dominator tree* D is a tree with root s in which each vertex v has $d(v)$ as its parent. Dominator trees can be computed in linear time [2, 7, 8, 23]. The problem of finding dominators has been extensively studied, as it occurs in several applications, including program optimization and code generation [12], constraint programming [34], circuit testing [4], theoretical biology [1], memory profiling [31], fault-tolerant computing [5, 6], connectivity and path-determination problems [16, 17, 19, 18, 25, 27, 28, 29], and the analysis of diffusion networks [24].

In particular, the dynamic dominator problem arises in various applications, such as data flow analysis and compilation [11, 15]. Moreover, the results of Italiano et al. [27] imply that dynamic dominators can be used for dynamically testing 2-vertex connectivity, and for maintaining the strong bridges and strong articulation points of digraphs. The decremental dominator problem appears in the computation of maximal 2-connected subgraphs in digraphs [25, 29, 13]. The problem of updating the dominator relation has been studied for a few decades (see, e.g., [3, 9, 11, 20, 33, 35, 36]). For the incremental dominator problem, there are algorithms that achieve total $O(mn)$ running time for processing a sequence of edge insertion in a flow graph with n vertices, where m is the number of edges after all insertions [3, 11, 20]. Moreover, they can answer dominance queries, i.e., whether a query vertex w dominates another query vertex v, in constant time. Prior to our work, to the best of our knowledge, no decremental algorithm with total running time better than $O(m^2)$ was known for general flow graphs. In the special case of *reducible* flow graphs (a class that includes acyclic flow graphs), Cicerone et al. [11] achieved an $O(mn)$ update bound for the decremental dominator problem. Both the incremental and the decremental algorithms of [11] require $O(n^2)$ space, as they maintain the transitive closure of the digraph.

Our algorithm is the first to improve the trivial $O(m^2)$ bound for the decremental dominator problem in *general* flow graphs. Specifically, our algorithm can process a sequence of edge deletions in a flow graph with n vertices and initially m edges in $O(mn \log n)$ time and $O(n^2 \log n)$ space, and after processing each deletion can answer dominance queries in constant time. For the special case of *reducible* flow graphs, we give an algorithm that matches the $O(mn)$ running time of Cicerone et al. while improving the space usage to $O(m + n)$. We remark that the reducible case is interesting for applications in program optimization since one notion of a "structured" program is that its flow graph is reducible. (The details about this result appear in the full paper.) Finally, we complement our results with a conditional lower bound, which suggests that it will be hard to substantially improve our update bounds. In particular, we prove that there is no incremental nor decremental algorithm for maintaining the dominator tree (or more generally, a dominance data structure) that has total update time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$) unless the OMv Conjecture [26] fails. The same lower bound applies to the extended decremental SCC problem. Unlike the update time, it is not clear that the $O(n^2 \log n)$ space used by our joint SCC-decomposition is near-optimal. We leave it as an open problem to improve this bound.

Further Notation and Terminology. For a given digraph $G = (V, E)$, we denote the set of vertices by $V(G) = V$ and the set of edges by $E(G) = E$. We let $|E| = m$ and $|V| = n$. Two vertices u and v are *strongly connected* in G if there is a path from u to v and a path from v to u. G is *strongly connected* if every vertex is reachable from every other vertex. The *strongly connected components* (SCCs) of G are its maximal strongly connected subgraphs. We denote by $G \setminus S$ (resp., $G \setminus e$)) the graph obtained after deleting a set S of vertices (resp., an edge e) from G. For a strongly connected graph H, we say that deleting an edge e *breaks* H, if $H \setminus e$ is not strongly connected.

An edge (resp., a vertex) of G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of SCCs. An edge e (resp., a vertex v) is a *separating edge* (resp., a *separating vertex*) for two vertices u and v if u and v are not strongly connected in $G \setminus e$ (resp., in $G \setminus v$). Two vertices u and v are 2-*edge connected* (2-*vertex connected*) if there are two edge-disjoint paths (internally vertex-disjoint paths) between u and v. We denote by G^{R} the *reverse graph* of G, i.e., the graph which has the same vertices as G and contains an edge $e^{\mathrm{R}} = (v, u)$ for every edge $e = (u, v)$ of G.

2 A Data Structure for Maintaining Joint SCC-Decompositions

For a given initial graph G, the *decremental SCC problem* asks us to maintain a data structure that allows edge deletions and can answer whether (arbitrary) pairs (u, v) of vertices are in the same SCC. The goal is to update the data structure as quickly as possible while answering queries in constant time. In this paper we present a data structure for the *extended decremental SCC problem* in which a query provides an additional vertex w and asks whether u and v are in the same SCC when w is deleted from G. We maintain this information under edge deletions, and our data structure relies on Łącki's *SCC-decomposition* [30] for doing so.

2.1 Review of Łącki's SCC Decomposition

An SCC-decomposition recursively partitions the graph G into smaller strongly connected subgraphs. This generates a rooted tree T, whose root r represents the entire graph, and where the subtree rooted at each node ϕ represents some vertex-induced strongly connected subgraph G_ϕ (we refer to vertices of T as nodes to distinguish T from G). Every non-leaf node ϕ is a vertex of G_ϕ, and the children of ϕ correspond to SCCs of $G_\phi \setminus \phi$. The concept was introduced by Łącki [30] and slightly extended by Chechik et al. [10]. We adopt the notation from [10].

▶ **Definition 1** (SCC-decomposition). Let $G = (V, E)$ be a strongly connected graph. An *SCC-decomposition* of G is a rooted tree T, whose nodes form a partition of V. For a node ϕ of T we define G_ϕ to be the subgraph of G induced by the union of all descendants of ϕ (including ϕ). Then, the following properties hold:

- Each internal node ϕ of T is a single-element set. (In this case, we sometimes abuse notation and assume that ϕ is the vertex itself.)

- Let ϕ be any internal node of T, and let $H_1, \ldots, H_t$ be the SCCs of $G_\phi \setminus \phi$. Then the node ϕ has t children $\phi_1, \ldots, \phi_t$, where $G_{\phi_i} = H_i$ for all $i \in \{1, \ldots, t\}$.

An SCC-decomposition of a graph G that is not strongly connected is a collection of SCC-decompositions of the SCCs of G. We say that T is a *partial* SCC-decomposition when the leaves of T are not required to be singletons.

Observe that for each node ϕ, the graph G_ϕ is strongly connected. Moreover, the subtree of T rooted at ϕ is an SCC-decomposition of G_ϕ. Also, for a leaf ϕ we have that $\phi = V(G_\phi)$. To build an SCC-decomposition T of a strongly connected graph G we pick an arbitrary vertex v, put it in the root of T, then recursively build SCC-decompositions of SCCs of $G \setminus \{v\}$ and make them the children of v in T. Note that since the choice of v is arbitrary, there are many ways to build an SCC-decomposition of the same graph. Łącki [30] (and Chechik et al. [10]) introduced a procedure Build-SCC-Decomposition(G, S) that takes as input a set of vertices S and returns a partial SCC-decomposition whose internal nodes are the vertices of S, i.e., these vertices are picked first and therefore appear at the top of the constructed tree. We refer to the vertices in S as internal nodes and the remaining nodes as external nodes. Note that all external nodes appear in the leaves of T, while internal nodes can be both leaves and non-leaves. This distinction is helpful when describing our algorithm.

Łącki [30] showed that the total initialization and update time under edge deletions of an SCC-decomposition is $O(m\gamma)$, where γ is the depth of the decomposition.

2.2 Towards a Joint SCC-Decomposition

Recall that the extended decremental SCC problem asks us to maintain under edge deletions a data structure for a graph G such that we can answer whether u and v are strongly connected in $G \setminus \{w\}$ when given $u, v, w \in V(G)$. A naive algorithm does this by maintaining n SCC-decompositions, each with a distinct vertex w as its root. The children of w in an SCC-decomposition that has w as its root are then exactly the SCCs of $G \setminus \{w\}$. Hence, u and v are in the same SCC if and only if they appear in the same subtree below w. The total update time of this data structure is however $O(mn^2)$, which is undesirable. With a more refined approach, we improve the time bound to $O(mn \log n)$.

Observe that the external nodes of a partial SCC-decomposition T produced by the procedure Build-SCC-Decomposition(G, S) exactly correspond to the SCCs of $G \setminus S$. This is true regardless of the order in which vertices from S are picked by the procedure. If two SCC-decompositions are built using the same set S, but with vertices being picked in a different order, then the nodes below S represent the same SCCs, which means that they can be shared by the two SCC-decompositions. Our algorithm is based on this observation. We essentially construct the n SCC-decompositions of the naive algorithm described above such that large parts of their subtrees are shared, and such that we do not need to maintain multiple copies of these subtrees. The idea is to partition the set S into two subsets S_1 and S_2 of equal size (we assume for simplicity that n is a power of 2), and then construct half of the SCC-decompositions with S_1 at the top and the other half with S_2 at the top. The procedure is repeated recursively on the top part of both halves. We refer to the bottom part, i.e., nodes that are not from S_1 and S_2, respectively, as the *extension* of the top part. Note that we eventually get a distinct vertex as the root of each of the n SCC-decompositions. The following definition formalizes the idea.

▶ **Definition 2** (Joint SCC-decomposition). A *joint SCC-decomposition* J is a recursive structure. It is either a regular SCC-decomposition T (the base case), or a pair of joint SCC-decompositions J_1, J_2 with the same set of internal nodes S and a shared set of external nodes

Input: A graph G and a set of vertices $S \subseteq V(G)$
Output: A balanced joint SCC-decomposition $J = (J_1, J_2, S, \Phi)$ of G on S.

```
if |S| = 1 then
    return T = Build-SCC-Decomposition(G, S).
end
Let S_1 and S_2 be the first and second half of S, respectively, and let Φ be an empty list.
foreach i ∈ {1, 2} do
    Compute J_i = Build-Joint-SCC-Decomposition(G, S_i).
    foreach external node φ of J_i do
        Compute T_φ = Build-SCC-Decomposition(G_φ, φ ∩ S).
        Add each external node of T_φ to Φ, if it is not already there.
    end
end
return J = (J_1, J_2, S, Φ)
```

Figure 1 Build-Joint-SCC-Decomposition(G,S).

Φ. In the second case we refer to J as the tuple (J_1, J_2, S, Φ). A joint SCC-decomposition $J = (J_1, J_2, S, \Phi)$ is *balanced* on S if it has one of the following two properties:

1. S is a singleton and J is a regular (partial) SCC-decomposition T with the vertex from S as root and no other internal nodes (the base case).
2. S can be partitioned into two equally sized halves S_1 and S_2, and J consists of two joint SCC-decompositions $J_1 = (J_{1,1}, J_{1,2}, S_1, \Phi_1)$ and $J_2 = (J_{2,1}, J_{2,2}, S_2, \Phi_2)$ that are balanced on S_1 and S_2, respectively. Also, each external node ϕ in Φ_1 and Φ_2 is extended with an associated SCC-decomposition T_ϕ for G_ϕ whose internal nodes are those of $\phi \cap S$. The combined set of external nodes of T_ϕ for all $\phi \in \Phi_1$ is equal to the combined set of external nodes of $T_{\phi'}$ for all $\phi' \in \Phi_2$, and these nodes are the external nodes Φ of J.

The procedure Build-Joint-SCC-Decomposition(G, S) describes how we build a balanced joint SCC-decomposition. G is the graph that we wish to decompose, and S is the set of vertices that we wish to place at the top. Initially S is the set of all vertices. The following lemma bounds the number of SCC-decompositions that make up a joint balanced SCC-decomposition. The lemma is proved by observing that the number of SCC-decompositions constructed by Build-Joint-SCC-Decomposition(G, S) is given by the recurrence $g(s) = 2g(s/2) + 2$ when $s > 1$ and $g(s) = 1$ otherwise, where $s = |S|$.

▶ **Lemma 3.** *A balanced joint SCC-decomposition for a graph G with n vertices consists of $O(n)$ SCC-decompositions.*

▶ **Lemma 4.** *Let $J = (J_1, J_2, S, \Phi)$ be a balanced joint SCC-decomposition of a graph G such that $S = V(G)$. Then the total number of nodes of J is $O(n \log n)$, where $n = |V(G)|$.*

Proof. The proof is by induction. Our induction hypothesis says that the total number of internal nodes of a balanced joint SCC-decomposition $J = (J_1, J_2, S, \Phi)$, counting not only S but also recursively the number of internal nodes of J_1 and J_2, is $|S| \cdot (1 + \log |S|)$.

In the base case, J is an SCC-decomposition with a single internal node, and the induction hypothesis is clearly satisfied. For the induction step we count separately the total number of internal nodes of $J_1 = (J_{1,1}, J_{1,2}, S_1, \Phi_1)$ and $J_2 = (J_{2,1}, J_{2,2}, S_2, \Phi_2)$, and add the number of internal nodes of the SCC-decompositions T_ϕ for $\phi \in \Phi_1$ and $\phi \in \Phi_2$, i.e., the extensions of J_1 and J_2 to S. Since $|S_1| = |S_2| = |S|/2$, it follows from the induction hypothesis that both J_1 and J_2 have $\frac{|S|}{2} \cdot \log |S|$ internal nodes in total. The internal nodes of T_ϕ for $\phi \in \Phi_1$ are exactly S_2, and the internal nodes of T_ϕ for $\phi \in \Phi_2$ are exactly S_1. Hence the number of internal nodes in the extensions are $|S_1| + |S_2| = |S|$. It follows that the total number of internal nodes of J is $|S| \cdot (1 + \log |S|)$ as desired.

It remains to count the external nodes of J. Note that external nodes of J_1 and J_2 correspond to internal nodes of J, i.e., they are roots of the SCC-decompositions that extend J_1 and J_2. Therefore there are at most as many external nodes inside the recursion as there are internal nodes in total. There are $O(n)$ external nodes in the extensions of J_1 and J_2 to S, and we conclude that the total number of nodes when $S = V(G)$ is at most $O(n \log n)$. ◀

Recall that an SCC-decomposition of depth γ can be initialized in time $O(m\gamma)$ [30]. Since the depth of an SCC-decomposition is at most the number of internal nodes plus one, and since Lemma 4 shows that the total number of nodes in a joint SCC-decomposition is $O(n \log n)$, it follows that the combined depth of all the SCC-decompositions that make up a joint SCC-decomposition is at most $O(n \log n)$, which proves the following lemma.

▶ **Lemma 5.** *The procedure* Build-Joint-SCC-Decomposition(G, S) *constructs a joint SCC-decomposition in time* $O(mn \log n)$*.*

To answer queries for the extended decremental SCC problem in constant time, we also construct and maintain an $n \times n$ matrix A such that $A[u, w]$ is the index of the SCC of $G \setminus \{w\}$ that contains u. Two vertices u and v are in the same SCC of $G \setminus \{w\}$ if and only if $A[u, w] = A[v, w]$. To avoid cluttering the pseudo-code we describe separately how A is maintained. In Build-Joint-SCC-Decomposition(G, S) we initialize A in the base case when we compute an SCC-decomposition T for a singleton $S = \{w\}$. Indeed, in this case w is the root of T, and the external nodes are exactly the SCCs of $G \setminus \{w\}$. Hence, for every vertex $u \in V(G) \setminus \{w\}$ we set $A[u, w]$ to the index of the SCC it is part of in $G \setminus \{w\}$.

Note that storing the matrix A takes space $O(n^2)$. The time spent initializing A is however dominated by the other work performed by the algorithm.

2.3 Deleting Edges from a Joint SCC-Decomposition

We next show how to maintain a joint SCC-decomposition under edge deletions. It is again instructive to consider the work performed by the naive algorithm that maintains n SCC-decompositions with distinct roots. If these are constructed as described in Section 2.2, then the SCC-decompositions will share many identical subtrees, and the work performed on these subtrees will be the same. In the joint SCC-decomposition such subtrees are shared, but otherwise the work performed is the same as the work performed for individual SCC-decompositions. We therefore use Łącki's algorithm [30] to delete edges from the individual SCC-decompositions, and we introduce a new procedure for handling the interface between the SCC-decompositions. We next briefly sketch Łącki's algorithm (see also [10, 30]).

Recall that each node ϕ of an SCC-decomposition T represents a strongly connected subgraph G_ϕ induced by the vertices in the subtree rooted at ϕ. If ϕ is an internal node of T, then the children of ϕ are the SCCs of $G_\phi \setminus \phi$. Łącki uses the following two operations to compactly represent edges among ϕ and its children.

▶ **Definition 6.** Let G be a graph. The *condensation* of G, denoted by Condense(G), is the graph obtained from G by contracting all its SCCs into single vertices. Let $v \in V(G)$. By Split(G, v) we denote the graph obtained from G by splitting v into two vertices: v_{in} and v_{out}. The in-edges of v are connected to v_{in} and the out-edges to v_{out}.

The two operations are often used together, and to simplify notation we use the shorthand $G_v^{\text{con}} = \text{Condense}(\text{Split}(G, v))$. The graph G_ϕ^{con} is stored with every internal node ϕ of the SCC-decomposition T. This introduces at most three copies of every vertex v of G: The two vertices v_{in} and v_{out} in G_v^{con}, and possibly a third vertex in the condensed graph of the parent of v in T. Moreover, every edge (u, v) appears in exactly one condensed graph, namely that of the lowest common ancestor of u and v in T, which we denote by $\text{LCA}(u, v)$. The combined space used for storing all the condensed graphs is thus $O(m + n)$.

To delete an edge (u, v), Łącki [30] locates $\phi = \text{LCA}(u, v)$, and deletes (u', v') from G_ϕ^{con}, where u' and v' are the vertices whose subtrees contain u and v. (He uses $O(m)$ space to store a pointer from every edge (u, v) to $\text{LCA}(u, v)$, enabling him to find the lowest common ancestor in constant time.) To preserve connectivity, he then checks whether u' and v' have non-zero out- and in-degrees, respectively, in G_ϕ^{con}. If this is not the case, then he repeatedly removes vertices with out- or in-degree zero and their adjacent edges from G_ϕ^{con}. All such vertices can be located, starting from u' and v', in time that is linear in the number of edges adjacent to the removed vertices. The corresponding children of ϕ are then moved up one level in T and are made siblings of ϕ. They are also inserted into $G_{par(\phi)}^{\text{con}}$, where $par(\phi)$ is the parent of ϕ, and their edges and the edges of ϕ in $G_{par(\phi)}^{\text{con}}$ are updated correspondingly. This can again be done in time linear in the number of edges in the original graph that are adjacent to vertices in the subtrees that are moved. The procedure is then repeated in $G_{par(\phi)}^{\text{con}}$. Since every vertex increases its level at most γ times, where γ is the initial depth of T, it follows that the total update time of the algorithm is at most $O(m\gamma)$.

We let Delete-Edge-from-SCC-decomposition(T, u, v) be the procedure for deleting an edge (u, v) from an SCC-decomposition T. We also denote the recursive procedure for moving nodes $\phi_1, \ldots, \phi_k$ from being children of ϕ to being siblings of ϕ in T after an edge (u, v) is deleted by Fix-SCC-decomposition$(T, u, v, \phi, \{\phi_1, \ldots, \phi_k\})$. Both procedures return the resulting SCC-decomposition, or a collection of SCC-decompositions in case the graph is not strongly connected. (The pseudo-code appears in the full version of the paper.)

In a joint SCC-decomposition, vertices and edges may appear in multiple nodes as part of smaller SCC-decompositions. We therefore need to find every occurence of the edge that we wish to delete. We introduce a procedure Delete-Edge(J, u, v) that does that by recursively searching through the nested joint SCC-decompositions and deleting (u, v) from the relevant SCC-decompositions. The procedure also handles the interface between SCC-decompositions. Note that deleting (u, v) from an SCC-decomposition T may cause the SCC corresponding to the root ϕ of T to break. The procedure Delete-Edge-from-SCC-decomposition(T, u, v) will in this case return a collection of SCC-decompositions $\{T_1, \ldots, T_k\}$, one for each new SCC. Suppose $J = (J_1, J_2, S, \Phi)$. If T extends J_1 (resp. J_2), then it is an SCC-decomposition of the subgraph G_ϕ associated with some external node ϕ of J_1 (resp. J_2). ϕ is then itself a leaf of an SCC-decomposition T' in J_1 (resp. J_2). Moreover, when the SCC corresponding to ϕ breaks, then this leaf must be split into multiple leaves of T', one for each new SCC. Note however that the levels in T' of the involved vertices do not change after the split. We therefore cannot charge the work performed when splitting ϕ to the analysis by Łącki [30].

Let $\phi_1, \ldots, \phi_k$ be the roots of the SCC-decompositions $T_1, \ldots, T_k$ that are created when the deletion of (u, v) breaks the SCC G_ϕ. As mentioned above, we need to replace ϕ in T' by $\phi_1, \ldots, \phi_k$, which means that $\phi_1, \ldots, \phi_k$ should replace ϕ in $G_{par(\phi)}^{\text{con}}$, where $par(\phi)$ is the parent of ϕ in T'. To efficiently reconnect $\phi_1, \ldots, \phi_k$ in $G_{par(\phi)}^{\text{con}}$ we identify the vertex ϕ_i whose associated graph G_{ϕ_i} has the most vertices, and we then scan through all the vertices in the other graphs $G_{\phi_1}, \ldots, G_{\phi_{i-1}}, G_{\phi_{i+1}}, \ldots, G_{\phi_k}$ and reconnect their adjacent edges in $G_{par(\phi)}^{\text{con}}$ when relevant. The work performed is exactly the same as when Łącki fixes an SCC-decomposition after an edge is removed. We can therefore call FIX-SCC-DECOMPOSITION(T',u,v,ϕ,$\{\phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \ldots, \phi_k\}$). Note that this makes ϕ_i take over the role of ϕ. Also note that we provide the procedure with the end-points u and v of the edge that was deleted, since u and v are used as starting points for the search for disconnected vertices when propagating the update further up the tree.

Finally, observe that splitting the leaf ϕ of the SCC-decomposition T' may propagate all the way to the root of T' and break the SCC corresponding to T'. We therefore use a recursive procedure, SPLIT-LEAF(J,u,v,ϕ,$\{\phi_1, \ldots, \phi_k\}$), to perform the split. Here u and v are again the end-points of the edge that was deleted.

▶ **Theorem 7.** *The total update time spent by* DELETE-EDGE(J, u, v) *in order to maintain a balanced joint SCC-decomposition under edge deletions is* $O(mn \log n)$*.*

Proof. We only sketch the proof, and refer to the full paper for additional details.

The time spent by DELETE-EDGE(J, u, v) consists of three parts: checking whether $\{u, v\} \subseteq V(G_\phi)$ for some $\phi \in \Phi$, the work performed while deleting edges from SCC-decompositions, and the work performed by SPLIT-LEAF($J_i, u, v, \phi, \{\phi_1, \ldots, \phi_k\}$), for $i \in \{1, 2\}$. To check in constant time whether $\{u, v\} \subseteq V(G_\phi)$, we maintain for each SCC-decomposition T an array on the vertices of the original graph G, such that $T\langle v\rangle =$ TRUE if v appears in T, and $T\langle v\rangle =$ FALSE otherwise. Storing these arrays takes up $O(n^2)$ space, and they are updated when the SCC of the root of an SCC-decomposition breaks.

Recall that Łącki [30] showed that the total initialization and update time of an SCC-decomposition is $O(m\gamma)$, where γ is the depth of the decomposition. By Lemma 4, the total number of nodes of the SCC-decompositions in J is $O(n \log n)$, and therefore the combined depth of the SCC-decompositions is also $O(n \log n)$. It follows that the time spent on DELETE-EDGE-FROM-SCC-DECOMPOSITION(T_ϕ, u, v) is bounded by $O(mn \log n)$.

It remains to analyze the time spent on SPLIT-LEAF($J_i, u, v, \phi, \{\phi_1, \ldots, \phi_k\}$). Recall that SPLIT-LEAF identifies the node ϕ_i that contains the most vertices from G, and then breaks off $\phi_1, \ldots, \phi_{i-1}, \phi_i, \ldots, \phi_k$ from ϕ. This means that ϕ is turned into ϕ_i, and that we do not scan through edges adjacent to vertices in ϕ_i. Since a split therefore moves vertices to new nodes of at most half the size, each vertex v can only be moved $O(\log n)$ times in one particular SCC-decomposition T by SPLIT-LEAF. Each move takes time proportional to the number of edges adjacent to v, so the total time spent splitting leaves of T is at most $O(m \log n)$. Since, by Lemma 3, there are only $O(n)$ SCC-decompositions in J, it follows that the total time spent splitting leaves is $O(mn \log n)$. Furthermore, the time spent by SPLIT-LEAF on fixing SCC-decompositions can be charged to the depth reduction of the vertices that are moved, and this part of the analysis is therefore the same as for DELETE-EDGE-FROM-SCC-DECOMPOSITION(T_ϕ, u, v). ◀

Recall that we also maintain a matrix A for answering queries, where $A[u, w]$ is the index of the SCC of $G \setminus \{w\}$ that contains u. We again update A when we make changes to the topmost SCC-decompositions that each only contain a single internal node, i.e., when such a root ϕ gets a new child, or when G_ϕ breaks into multiple SCCs. The time spent updating A is dominated by the rest of the work that is performed by our algorithm, where we scan through all edges adjacent to vertices whose SCC is changed.

As described briefly in Section 2.3, Łącki's SCC-decomposition can be implemented such that is uses $O(m + n)$ space [30]. Since a balanced joint SCC-decomposition consists of $O(n)$ SCC-decompositions (Lemma 3), it follows that a naive implementation of our data structure uses $O(mn)$ space. In the full version of the paper we show how to obtain an alternative bound of $O(n^2 \log n)$. Doing so requires two observations:

- After an edge (u', v') is deleted from a condensed graph G^{con}_{ϕ}, the vertex u' has a path to ϕ_{in} if and only if u' has non-zero out-degree, and there is a path from ϕ_{out} to v' if and only if v' has non-zero in-degree. Instead of storing the edges of the condensed graphs we therefore store the in- and out-degrees of the vertices. To visit all neighbors of a vertex u' in G^{con}_{ϕ}, we then collect from the original graph G all edges adjacent to vertices in the subgraph of u', and we check for each edge whether the other end-point is part of G^{con}_{ϕ} and which vertex of G^{con}_{ϕ} it goes to. To do so we store pointers between the vertices of G and the vertices of the condensed graphs that they are part of. Since a joint SCC-decomposition contains $O(n \log n)$ nodes this takes $O(n^2 \log n)$ space.
- Since a joint SCC-decomposition contains $O(n \log n)$ nodes in total, we have time to visit all the nodes of an SCC-decomposition T when searching for the lowest common ancestor of two vertices u and v. This is done in a bottom-up fashion. We therefore do not need to store a pointer from every edge (u, v) to $\text{LCA}(u, v)$.

3 Applications

In this section we exploit the decremental joint SCC-decomposition to design decremental algorithms for various connectivity notions defined with respect to vertex or edge failures.

Maintaining Decrementally the Dominator Tree. We show how to maintain a dominator tree D of a flow graph G, rooted at a starting vertex s. We denote by $d(v)$ the parent of v in D. We first produce a flow graph G_s from G by adding an edge from each vertex $v \in V \setminus s$ to s. The addition of those edges has the following property. If a vertex u is not strongly connected to s in G_s then there is no path from s to u in G. Conversely, if a vertex u is not strongly connected to s in $G_s \setminus v$, while s and u are strongly connected in G_s, then all paths from s to u in G contain v. That is, v is a dominator of u in G.

We maintain decrementally a joint SCC-decomposition of G_s in a total of $O(mn \log n)$ time and $O(n^2 \log n)$ space. Therefore, for each vertex v we maintain the SCCs in $G \setminus v$. Let $v \neq s$: after the SCC containing s in $G \setminus v$ breaks, all the vertices that are not in the new SCC that contains s are dominated from v in G. We can report the newly dominated vertices from v in G in time proportional to their number. Therefore, after each edge deletion we need to process a batch $\mathcal{N}$ of incoming new dominance relations $N(v) = \{u_1, \ldots, u_k\}$, where $u_1, \ldots, u_k$ are dominated by v in G. We can process a batch of updates $\mathcal{N}$ in two phases as follows. For each vertex $u \neq s$ we keep a counter $\mathit{depth}(u)$ of the number of vertices that dominate u. For each dominance relation $N(v) \in \mathcal{N}$, we increase $\mathit{depth}(u)$ for each $u \in N(v)$. After this first phase ends, all vertices have updated counters. Then the new parent of each vertex u in D is the vertex with the largest counter among $d(u)$ (i.e., the parent of u in D before the edge insertion) and all vertices v such that $u \in N(v)$ and $N(v) \in \mathcal{N}$.

Now we bound the total time required to maintain the dominator tree. The running time of the above procedure, during the whole sequence of deletions, is bounded by the total size of all the sets $N(v)$. Note that any vertex can appear in a specific $N(v)$ set at most once during the deletion sequence. Hence, the total size of all the sets $N(v)$ is $O(n^2)$.

▶ **Lemma 8.** *The dominator tree of a directed graph G with start vertex s can be maintained decrementally in $O(mn \log n)$ total update time and $O(n^2 \log n)$ space, where m is the number of edges in the initial graph and n is the number of vertices.*

Maintaining Decrementally the Strong Bridges of the Graph. Let G be strongly connected: its strong bridges can be computed efficiently from a dominator tree D of G and a dominator tree D^R of G^R, both rooted at the same start vertex s. We present a randomized algorithm that maintains such a pair of dominator trees in each SCC of a digraph in $O(mn \log n)$ total expected time, and $O(n^2 \log n)$ space. This allows us to maintain the set of strong bridges of a digraph in the same (expected) running time and space.

Maintaining Decrementally the 2-Edge-Connected Components. In this section we show how to maintain the 2-edge-connected components of directed graph. Two vertices w and z are 2-edge-connected if and only if there is no edge e such that w and z are not strongly connected in $G \setminus e$. A 2-edge-connected component is a maximal subset $B \subseteq V$, such that w and z are 2-edge-connected, for all $z, w \in B$. Therefore, a simple-minded algorithm for computing the 2-edge-connected components is the following. We start with the trivial partition $\mathcal{P}$ of the vertices that is equal to the set of SCCs of the graph. For every strong bridge e, we compute the SCCs $C_1, \dots, C_k$ of $G \setminus e$ and we refine the maintained partition $\mathcal{P}$ according to the partition induced by the SCCs $C_1, \dots, C_k$. After performing all refinements on $\mathcal{P}$ two vertices are in the same set if and only if we did not find an edge that separates them, which is exactly the definition of 2-edge-connected components.

Our algorithm is a dynamic version of the aforementioned simple-minded algorithm. That is, we maintain the SCCs of $G \setminus e$, for each strong bridge e, and refine the maintained partition $\mathcal{P}$ whenever we identify that $\mathcal{P}$ no longer contains the 2-vertex-connected components of G. We do this as follows. Assume that a component $C \in \mathcal{P}$ contains vertices in different SCCs of $G \setminus e$, for some e. Let $C_1, C_2, \dots, C_k$ be the SCCs in $G \setminus e$. We replace C by $\{C \cap C_1\}, \dots, \{C \cap C_k\}$. These refinements can be easily performed in $O(n)$ time, and therefore we spend total time $O(n^2)$ for all refinements throughout the algorithm.

In order to make our algorithm efficient we need to specify how to detect whether two 2-edge-connected vertices appear in different SCCs in $G \setminus e$, for some edge e. Whenever an SCC C in $G \setminus e$, breaks into k SCCs $C_1, \dots, C_k$, for all SCCs C_i except the largest one we examine whether the components $C \in \mathcal{P}$ containing subsets of vertices of C_i are entirely contained in C_i. We develop machinery that allows us to list all vertices in the resulting SCCs $C_1, \dots, C_k$ except the largest one, in time proportional to their number. The details can be found in the full paper. Each vertex can appear at most $\log n$ times in an SCC of $G \setminus e$, for some strong bridge e, that is not the largest after a big SCC breaks. This implies that we spend $O(n \log n)$ time for each graph $G \setminus e$ on testing whether an edge deletion leaves two (previously) 2-edge-connected vertices in different SCCs in $G \setminus e$, for some edge e. We show that at most $O(n)$ strong bridges can appear throughout any sequence of edge deletions. Thus we spend $O(n^2 \log n)$ time in total.

▶ **Lemma 9.** *The 2-edge-connected components of a digraph G can be maintained decrementally in $O(mn \log n)$ total expected time against an oblivious adversary, using $O(n^2 \log n)$ space, where m is the number of edges in the initial graph and n is the number of vertices.*

4 Conditional Lower Bound

In the following we give a conditional lower bound for the partially dynamic dominator tree problem. We show that there is no incremental nor decremental algorithm for maintaining the dominator tree that has total update time $O((mn)^{1-\epsilon})$ (for some constant $\epsilon > 0$) unless the OMv Conjecture [26] fails. This also holds for algorithms that do not explicitly maintain the tree, but are able to answer parent-queries. Formally, we prove the following statement.

▶ **Theorem 10.** *For any constant $\delta \in (0, 1/2]$ and any n and $m = \Theta(n^{1/(1-\delta)})$, there is no algorithm for maintaining a dominator tree under edge deletions/insertions allowing queries* *of the form "is x the parent of y in the dominator tree" that uses polynomial preprocessing time, total update time $u(m,n) = (mn)^{1-\epsilon}$ and query time $q(m) = m^{\delta-\epsilon}$ for some constant $\epsilon > 0$, unless the* OMv *conjecture fails.*

Under this conditional lower bound, the running time of our algorithm is optimal up to sub-polynomial factors. We give the reduction for the decremental version of the problem. Hardness of the incremental version follows analogously.

In the online Boolean matrix-vector problem we are first given a Boolean $n \times n$ matrix M to preprocess. After the preprocessing, we are given a sequence of n-dimensional Boolean vectors $v^{(1)}, \ldots, v^{(n)}$ one by one. For each $1 \leq t \leq n$, we have to return the result of the matrix-vector multiplication $Mv^{(t)}$ before we are allowed to see the next vector $v^{(t+1)}$. The OMv Conjecture states that there is no algorithm that computes each matrix-vector product correctly (with high probability) and in total spends time $O(n^{3-\epsilon})$ for some constant $\epsilon > 0$.

We will not use the OMv problem directly as the starting point of our reduction. Instead we consider the following γ-OuMv problem (for a fixed $\gamma > 0$) and parameters n_1, n_2, and n_3 such that $n_1 = \lfloor n_2^\gamma \rfloor$: We are first given a Boolean $n_1 \times n_2$ matrix M to preprocess. After the preprocessing, we are given a sequence of pairs of n_1-dimensional Boolean vectors $(u^{(1)}, v^{(1)}), \ldots, (u^{(n_3)}, v^{(n_3)})$ one by one. For each $1 \leq t \leq n_3$, we have to return the result of the Boolean vector-matrix-vector multiplication $(u^{(t)})^\intercal M v^{(t)}$ before we see the next pair of vectors $(u^{(t+1)}, v^{(t+1)})$. It has been shown [26] that under the OMv Conjecture, there is no algorithm for this problem with polynomial preprocessing time and total processing time $O(n_1^{1-\epsilon_1} n_2^{1-\epsilon_2} n_3^{1-\epsilon_3})$ such that all ϵ_i are ≥ 0 and at least one ϵ_i is a constant > 0.

We now give the reduction from the γ-OuMv problem with $\gamma = \delta/(1-\delta)$ to the decremental dominator tree problem. In the following we denote by v_i the i-th entry of a vector i and by $M_{i,j}$ the entry at row i and column j of a matrix M.

Consider an instance of the γ-OuMv problem with parameters $n_1 = m^{1-\delta}$, $n_2 = m^\delta$, and $n_3 = m^{1-\delta}$. We preprocess the matrix M by constructing a graph $G^{(0)}$ with the set of vertices $V = \{s, x_1, \ldots, x_{n_3}, x_{n_3+1}, y_1, \ldots, y_{n_1}, z_1, \ldots, z_{n_2}\}$ and the following edges: (1) an edge (s, x_1), and, for every $1 \leq t \leq n_3$, an edge (x_t, x_{t+1}), (2) for every $1 \leq j \leq n_2$, an edge (t, z_j), (3) for every $1 \leq t \leq n_3$ and every $1 \leq i \leq n_1$, an edge (x_t, y_i), and (4) for every $1 \leq i \leq n_1$ and every $1 \leq j \leq n_2$, an edge (y_i, z_j) if and only if $M_{i,j} = 1$.

Whenever the algorithm is given the next pair of vectors $(u^{(t)}, v^{(t)})$, we first create a graph $G^{(t)}$ by performing the following edge deletions in $G^{(t-1)}$: If $t \geq 2$, we first delete all outgoing edges of x_{t-1}, except the one to x_t. Then (for any value of t), for every i such that $u_i^{(t)} = 0$ we delete the edge from x_t to y_i. Thus, for every $1 \leq i \leq n_1$, there will be an edge from x_t to y_i in $G^{(t)}$ if and only if $u_i^{(t)} = 1$. Having created $G^{(t)}$, we now, for every j such that $v_j^{(t)} = 1$, check whether x_t is the parent of z_j in the dominator tree. If this is the case for at least one j we return that $(u^{(t)})^\intercal M v^{(t)}$ is 1, otherwise we return 0.

The correctness of our reduction follows from the following lemma.

▶ **Lemma 11.** *For every $1 \leq t \leq n$, the j-th entry of $(u^{(t)})^\intercal M$ is 1 if and only if x_t is the immediate dominator of z_j in $G^{(t)}$*

Note that $(u^{(t)})^\intercal M v^{(t)}$ is 1 if and only if there is a j such that both the j-th entry of $u^{(t)} M$ as well as the j-th entry of $v^{(t)}$ are 1. Furthermore, x_t is the parent of z_j in the dominator tree if and only if x_t is an immediate dominator of z_j in the current graph. Therefore the lemma establishes the correctness of the reduction.

The initial graph $G^{(0)}$ has $n := \Theta(n_1 + n_2 + n_3) = \Theta(m^\delta + m^{1-\delta}) = \Theta(m^{1-\delta})$ vertices and $\Theta(n_1 n_2 + n_2 n_3) = \Theta(m)$ edges. The total number of parent-queries is $O(n_1 n_3) = m^{2(1-\delta)}$. Suppose the total update time of the decremental dominator tree algorithm is $O(u(m,n)) =$

$(mn)^{1-\epsilon}$ and its query time is $O(q(m)) = m^{\delta-\epsilon}$. Using the reduction above, we can thus solve the γ-OuMv problem for the parameters n_1, n_2, n_3 with polynomial preprocessing time and total update time $O(u(m,n)+m^{2(1-\delta)}q(m)) = O(u(m,m^{1-\delta})+m^{2(1-\delta)}q(m)) = O(m^{2-\delta-\epsilon})$. Since $n_1 n_2 n_3 = m^{2-\delta}$, this means we would get an algorithm for the γ-OuMv problem with polynomial preprocessing time and total update time $O(n_1^{1-\epsilon_1} n_2^{1-\epsilon_2} n_3^{1-\epsilon_3})$ where at least one ϵ_i is a constant > 0. This contradicts the OMv Conjecture.

References

1. S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004. `doi:10.1016/j.jtbi.2004.05.009`.
2. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–21132, 1999. `doi:10.1137/S0097539797317263`.
3. S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.
4. M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proc. of the 19th IEEE VLSI Test Symposium*, pages 124–130, 2001. `doi:10.1109/VTS.2001.923428`.
5. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In *Proc. of the 29th Int'l. Symposium on Distributed Computing (DISC)*, pages 528–543, 2015. `doi:10.1007/978-3-662-48653-5_35`.
6. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. of the 48th ACM Symposium on Theory of Computing (STOC)*, pages 509–518, 2016. `doi:10.1145/2897518.2897648`.
7. A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008. `doi:10.1137/070693217`.
8. A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296, 1998. `doi:10.1145/295656.295663`.
9. M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 274–284, 1988. `doi:10.1145/73560.73584`.
10. S. Chechik, T. D. Hansen, G. F. Italiano, J. Lacki, and N. Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{\mathrm{O}}(\mathrm{m}\sqrt{\mathrm{n}})$ total update time. In *Proc. of the 57th IEEE Symposium on Foundations of Computer Science(FOCS)*, pages 315–324, 2016. `doi:10.1109/FOCS.2016.42`.
11. S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theorical Computer Science*, 203:69–90, August 1998. `doi:10.1016/S0304-3975(97)00288-0`.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. `doi:10.1145/115372.115320`.
13. W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-connectivity in directed graphs: An experimental study. In *Proc. of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 173–187, 2015. `doi:10.1137/1.9781611973754.15`.

14 D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook, 2nd Edition, Vol. 1*, pages 9.1–9.28. CRC Press, 2009.

15 K. Gargi. A sparse algorithm for predicated global value numbering. *SIGPLAN Not.*, 37(5):45–56, 2002. doi:10.1145/543552.512536.

16 L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. of the 37th Int'l. Coll. on Automata, Languages, and Programming (ICALP)*, pages 738–749, 2010. doi:10.1007/978-3-642-14165-2_62.

17 L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. of the 19th European Symposium on Algorithms (ESA)*, pages 13–24, 2011. doi:10.1007/978-3-642-23719-5_2.

18 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. of the 42nd Int'l. Coll. on Automata, Languages, and Programming (ICALP)*, pages 605–616, 2015. doi:10.1007/978-3-662-47672-7_49.

19 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016. doi:10.1145/2968448.

20 L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. of the 20th European Symposium on Algorithms (ESA)*, pages 491–502, 2012. doi:10.1007/978-3-642-33090-2_43.

21 L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental strong connectivity and 2-connectivity in directed graphs. Manuscript, 2017.

22 L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *Proc. of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1880–1899, 2017. doi:10.1137/1.9781611974782.123.

23 L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 869–878, 2004.

24 M. Gomez-Rodriguez, L. Song, N. Du, H. Zha, and B. Schölkopf. Influence estimation and maximization in continuous-time diffusion networks. *ACM Transactions on Information Systems*, 34(2):9:1–9:33, 2016. doi:10.1145/2824253.

25 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. of the 42nd Int'l. Coll. on Automata, Languages, and Programming (ICALP)*, pages 713–724, 2015. doi:10.1007/978-3-662-47672-7_58.

26 M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. of the 47th ACM Symposium on Theory of Computing (STOC)*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

27 G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. doi:10.1016/j.tcs.2011.11.011.

28 R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO – Theoretical Informatics and Applications*, 49(2):93–119, 2015. doi:10.1051/ita/2015001.

29 R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016. doi:10.1016/j.dam.2015.10.001.

30 J. Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 9(3):27, 2013. doi:10.1145/2483699.2483707.

31 E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proc. of the 16th ACM SIGKDD Int'l. Con. on Knowledge Discovery and Data Mining (KDD)*, pages 115–124, 2010. doi:10.1145/1835804.1835822.

32 M. Mihalák, P. Uznański, and P. Yordanov. Prime factorization of the Kirchhoff polynomial: Compact enumeration of arborescences. In *Proc. of the SIAM Analytic Algorithmics and Combinatorics (ANALCO)*, pages 93–105, 2016. `doi:10.1137/1.9781611974324.10`.

33 K. Patakakis, L. Georgiadis, and V. A. Tatsis. Dynamic dominators in practice. In *Proc. of the 16th Panhellenic Conference on Informatics (PCI)*, pages 100–104, 2011. `doi:10.1109/PCI.2011.28`.

34 L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. of the 8th International Conference on Practical Aspects of Declarative Languages (PADL)*, pages 73–87, 2006. `doi:10.1007/11603023_6`.

35 G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 287–296, 1994. `doi:10.1145/174675.177905`.

36 V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997. `doi:10.1145/202529.202531`.

37 R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.