Incremental 2-Edge-Connectivity in Directed Graphs*

Loukas Georgiadis[1], Giuseppe F. Italiano[2], and Nikos Parotsidis[3]

1 **University of Ioannina, Ioannina, Greece**
loukas@cs.uoi.gr
2 **University of Rome Tor Vergata, Rome, Italy**
giuseppe.italiano@uniroma2.it
3 **University of Rome Tor Vergata, Rome, Italy**
nikos.parotsidis@uniroma2.it

Abstract

We present an algorithm that can update the 2-edge-connected blocks of a directed graph with n vertices through a sequence of m edge insertions in a total of $O(mn)$ time. After each insertion, we can answer the following queries in asymptotically optimal time:

- Test in constant time if two query vertices v and w are 2-edge-connected. Moreover, if v and w are not 2-edge-connected, we can produce in constant time a "witness" of this property, by exhibiting an edge that is contained in all paths from v to w or in all paths from w to v.
- Report in $O(n)$ time all the 2-edge-connected blocks of G.

This is the first dynamic algorithm for 2-connectivity problems on directed graphs, and it matches the best known bounds for simpler problems, such as incremental transitive closure.

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 2-edge connectivity on directed graphs; dynamic graph algorithms; incremental algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2016.49

1 Introduction

A dynamic graph algorithm aims at updating efficiently the solution of a problem after an update, such as an edge insertion or an edge deletion, faster than recomputing it from scratch. A problem is said to be *fully dynamic* if the update operations include both insertions and deletions of edges, and it is said to be *partially dynamic* if only one type of update, either insertions or deletions, is allowed. More specifically, a problem is said to be *incremental* (resp., *decremental*) if only insertions (resp., deletions) are allowed. In this paper, we present new incremental algorithms for 2-edge connectivity problems on directed graphs (digraphs). Before defining the problem, we first review some definitions. Let $G = (V, E)$ be a digraph. G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* (in short *SCC's*) of G are its maximal strongly connected subgraphs. Vertices $u, v \in V$ are *strongly connected* if they are in the same SCC of G. An edge of G is a *strong bridge* if its removal increases the number of SCC's. Let G be strongly connected: G is 2-*edge-connected* if it has no strong bridges. The 2-*edge-connected components* of G are its maximal 2-edge-connected subgraphs. Vertices $u, v \in V$ are said to

* A full version of the paper is available at http://arxiv.org/abs/1607.07073.

43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016).
Editors: Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi;
Article No. 49; pp. 49:1–49:15

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

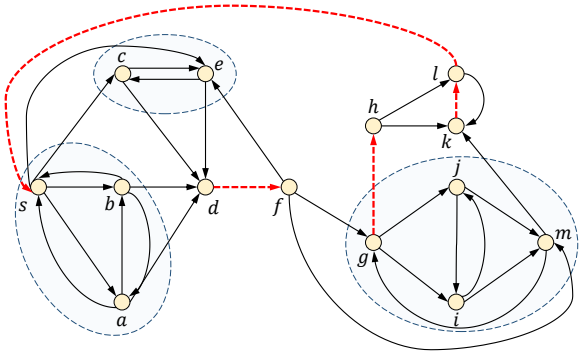

Figure 1 The 2-edge-connected blocks of a digraph G. Strong bridges of G are shown red and dashed. (Better viewed in color.)

be 2-*edge-connected*, denoted by $u \leftrightarrow_{2e} v$, if there are two edge-disjoint directed paths from u to v and two edge-disjoint directed paths from v to u. A 2-*edge-connected block* of a digraph $G = (V, E)$ is a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ for all $u, v \in B$ (see Figure 1).

We remark that in digraphs 2-connectivity has a much richer and more complicated structure than in undirected graphs. To see this, observe that, while in undirected graphs blocks are exactly the same as components, in digraphs there is a substantial difference between those two notions. In particular, two vertices that are 2-edge-connected (i.e., in the same 2-edge-connected block) may lie in different 2-edge-connected components (e.g., vertices i and j in Figure 1, each of them being in a 2-edge-connected component by itself). As a result, 2-connectivity problems on digraphs appear to be much harder than on undirected graphs. For undirected graphs it has been known for over 40 years how to compute 2-edge- and 2-vertex- connected components in linear time [33]. For digraphs, however, only $O(mn)$ algorithms were known (see e.g., [25, 26, 28, 30]). It was shown only recently how to compute the 2-edge- and 2-vertex- connected blocks in linear time [13, 14], and the best current bound for computing the 2-edge- and the 2-vertex- connected components is $O(n^2)$ [18].

Our Results. We initiate the study of the dynamic maintenance of 2-edge-connectivity relationships in directed graphs. We present an algorithm that can update the 2-edge-connected blocks of a digraph G with n vertices through a sequence of m edge insertions in a total of $O(mn)$ time. After each insertion, we can answer the following queries in asymptotically optimal time:

- Test in constant time if two query vertices v and w are 2-edge-connected. Moreover, if v and w are not 2-edge-connected, we can produce in constant time a "witness" of this property, by exhibiting an edge that is contained in all paths from v to w or in all paths from w to v.
- Report in $O(n)$ time all the 2-edge-connected blocks of G.

Ours is the first dynamic algorithm for 2-connectivity problems on digraphs, and it matches the best known bounds for simpler problems, such as incremental transitive closure [23]. This is a substantial improvement over the $O(m^2)$ simple-minded algorithm, which recomputes the 2-edge-connected blocks from scratch after each edge insertion.

Related Work. Many efficient algorithms for several dynamic graph problems have been proposed in the literature, including dynamic connectivity [20, 22, 31, 37], minimum spanning trees [8, 11, 21, 22], edge/vertex connectivity [8, 22] on undirected graphs, and transitive closure [7, 19, 27] and shortest paths [6, 27, 38] on digraphs. Once again, dynamic problems on digraphs appear to be harder than on undirected graphs. Indeed, most of the dynamic algorithms on undirected graphs have polylog update bounds, while dynamic algorithms on digraphs have higher polynomial update bounds. The hardness of dynamic algorithms on digraphs has been recently supported also by conditional lower bounds [1].

Our Techniques. Known algorithms for computing the 2-edge-connected blocks of a digraph G [13, 16] hinge on properties that seem very difficult to dynamize. The algorithm in [13] uses very complicated data structures based on 2-level auxiliary graphs. The loop nesting forests used in [16] depends heavily on an underlying dfs tree of the digraph, and the incremental maintenance of dfs trees on general digraphs is still an open problem (incremental algorithms are known only for the special case of DAGs [10]). Despite those inherent difficulties, we find a way to bypass loop nesting forests by suitably combining the approaches in [13, 16] in a novel framework, which is amenable to dynamic implementations. Another complication is that, although our problem is incremental, strong bridges may not only be deleted but also added (when a new SCC is formed). As a result, our data structures undergo a fully dynamic repertoire of updates, which is known to be harder. By organizing carefully those updates, we are still able to obtain the desired bounds. For lack of space, some technical details and proofs are omitted from this extended abstract and will be given in the full paper.

2 Dominator trees and 2-edge-connected blocks

Given a rooted tree, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v). Given a digraph $G = (V, E)$, and a set of vertices $S \subseteq V$, we denote by $G[S]$ the subgraph induced by S. We introduce next some of the building blocks of our new incremental algorithm.

Flow graphs, dominators, and bridges. A *flow graph* is a digraph with a distinguished *start vertex* s such that every vertex is reachable from s. Let $G = (V, E)$ be a strongly connected graph. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is obtained by reversing the direction of all edges. Let s be a fixed but arbitrary start vertex of a strongly connected digraph G. Since G is strongly connected, all vertices are reachable from s and reach s, so we can view both G and G^R as flow graphs with start vertex s. To avoid ambiguities, throughout the paper we will denote those flow graphs respectively by G_s and G_s^R. Vertex u is a *dominator* of vertex v (u *dominates* v) in G_s if every path from s to v in G_s contains u. The dominator relation can be represented by a tree D rooted at s, the *dominator tree* of G_s: u dominates v if and only if u is an ancestor of v in D. For any $v \neq s$, we denote by $d(v)$ the parent of v in D. Similarly, we can define the dominator relation in the flow graph G_s^R, and let D^R denote the dominator tree of G_s^R, and $d^R(v)$ the parent of v in D^R. Dominators and dominator trees can be computed in linear time [2, 5, 9, 12]. An edge (u, v) is a *bridge* of a flow graph G_s if all paths from s to v include (u, v).[1] Let s be an arbitrary start vertex of G.

[1] Throughout the paper, to avoid confusion we use consistently the term *bridge* to refer to a bridge of a flow graph and the term *strong bridge* to refer to a strong bridge in the original graph.

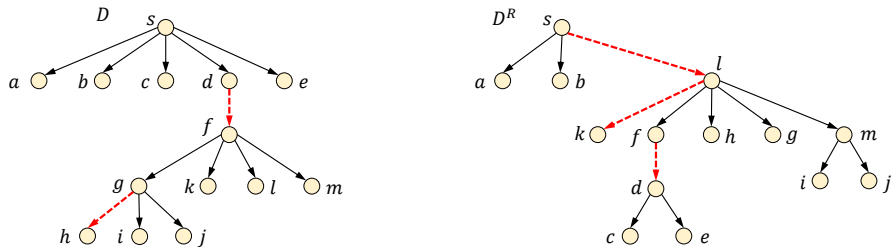

Figure 2 The dominator trees of flow graphs G_s and G_s^R. Strong bridges of G are shown red and dashed. (Better viewed in color.)

As shown in [24], an edge $e = (u, v)$ is strong bridge of G if and only if it is either a bridge of G_s or a bridge of G_s^R.

As a consequence, all the strong bridges of G can be obtained from the bridges of the flow graphs G_s and G_s^R, and thus there can be at most $2(n-1)$ strong bridges overall. Figure 2 illustrates the dominator trees D and D^R of the flow graphs G_s and G_s^R that correspond to the strongly connected digraph G of Figure 1. After deleting from the dominator trees D and D^R respectively the bridges of G_s and G_s^R, we obtain the *bridge decomposition* of D and D^R into forests $\mathcal{D}$ and $\mathcal{D}^R$. Throughout the paper, we denote by D_u (resp., D_u^R) the tree in $\mathcal{D}$ (resp., $\mathcal{D}^R$) containing vertex u, and by r_u (resp., r_u^R) the root of D_u (resp., D_u^R). The following lemma from [13] holds for a flow graph G_s of a strongly connected digraph G (and hence also for the flow graph G_s^R of G^R).

▶ **Lemma 1** ([13]). *Let G be a strongly connected digraph and let (u, v) be a strong bridge of G. Also, let D be the dominator tree of the flow graph G_s, for an arbitrary start vertex s. Suppose $u = d(v)$. Let w be any vertex that is not a descendant of v in D. Then there is path from w to v in G that does not contain any proper descendant of v in D. Moreover, all simple paths in G from w to any descendant of v in D must contain the edge $(d(v), v)$.*

Lemma 1 gives an initial partition of the vertices of G into subsets that contain the 2-edge-connected blocks of G. That is, for any two vertices u and v, we have $u \leftrightarrow_{2e} v$ only if u and v are in the same trees in the forests $\mathcal{D}$ and $\mathcal{D}^R$ (i.e., $r_u = r_v$ and $r_u^R = r_v^R$).

Loop nesting forests and bridge-dominated components. Let G be a digraph. A *loop nesting forest* represents a hierarchy of strongly connected subgraphs of G [35], defined with respect to a dfs tree T of G, as follows. For any vertex u, $loop(u)$ is the set of all descendants x of u in T such that there is a path from x to u in G containing only descendants of u in T. Any two vertices in $loop(u)$ reach each other. Therefore, $loop(u)$ induces a strongly connected subgraph of G; it is the unique maximal set of descendants of u in T (that includes u) that does so. The $loop(u)$ sets form a laminar family of subsets of V: for any two vertices u and v, $loop(u)$ and $loop(v)$ are either disjoint or nested. The *loop nesting forest* H of G, with respect to T, is the forest in which the parent of any vertex v, denoted by $h(v)$, is the nearest proper ancestor u of v in T such that $v \in loop(u)$ if there is such a vertex u, and null otherwise. Then $loop(u)$ is the set of all descendants of vertex u in H, which we will also denote as $H(u)$ (the subtree of H rooted at vertex u). A loop nesting forest can be computed in linear time [5, 35]. Since we deal with strongly connected digraphs, each vertex is contained in a loop, so H is a tree. Therefore, we will refer to H as the *loop nesting tree* of G. Let $e = (u, v)$ be a bridge of the flow graph G_s, and let $G[D(v)]$ denote the

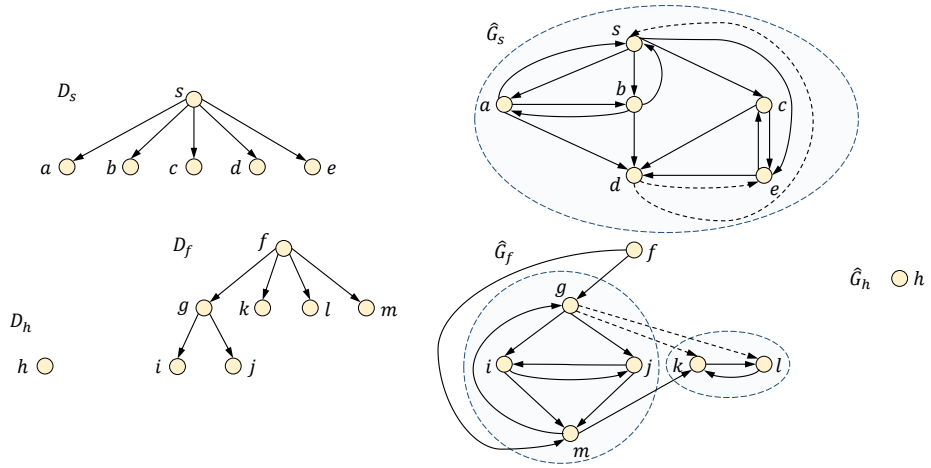

Figure 3 The bridge decomposition of the dominator tree D of Figure 2, the corresponding auxiliary graphs $\widehat{G}_r$ (the auxiliary edges are shown dashed) and their SCC's shown encircled.

subgraph induced by the vertices in $D(v)$. Let C be an SCC of $G[D(v)]$: we say that C is an *e-dominated component* of G. We also say that $C \subseteq V$ is a *bridge-dominated component* if it is an e-dominated component for some bridge e: it can be shown that bridge-dominated components form a laminar family. Let $e = (u, v)$ be a bridge of G_s, and let w be a vertex in $D(v)$ such that $h(w) \notin D(v)$. As shown in [16], $H(w)$ induces an SCC in $G[D(v)]$, and thus it is an e-dominated component.

Bridge decomposition and auxiliary graphs. Now we define a notion of *auxiliary graphs* that play a key role in our approach. Auxiliary graphs were defined in [13] to decompose the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original 2-edge-connected blocks of G. Unfortunately, the auxiliary graphs of [13] are not suitable for our purposes, and we need a slightly different definition. For each root r of a tree in the bridge decomposition $\mathcal{D}$ we define the *auxiliary graph* $\widehat{G}_r = (V_r, E_r)$ *of* r as follows. The vertex set V_r of $\widehat{G}_r$ consists of all the vertices in D_r. The edge set E_r contains all the edges of G among the vertices of V_r, referred to as *ordinary* edges, and a set of *auxiliary* edges, which are obtained by contracting vertices in $V \setminus V_r$, as follows. Let v be a vertex in V_r that has a child w in $V \setminus V_r$. Note that (v, w) is a bridge and w is a root in the bridge decomposition $\mathcal{D}$ of D. For each such child w of v, we contract w and all its descendants in D into v. Figure 3 shows the bridge decomposition of the dominator tree D and the corresponding auxiliary graphs. Differently from [13], our auxiliary graphs do not preserve the 2-edge-connected blocks of G. Note that each vertex appears exactly in one auxiliary graph. Furthermore, each original edge corresponds to at most one auxiliary edge. Therefore, the total number of vertices in all auxiliary graphs is n, and the total number of edges is at most m. We use the term *auxiliary components* to refer to the SCC's of the auxiliary graphs.

▶ **Lemma 2.** *All the auxiliary graphs of a flow graph G_s can be computed in linear time.*

A new algorithm for 2-edge-connected blocks. We next sketch a new linear-time algorithm to compute the 2-edge-connected blocks that combines ideas from [13] and [16] and that will be useful for our incremental algorithm. We refer to this algorithm as the 2ECB labeling algorithm. Similarly to [16], our algorithm assigns a label to each vertex, so that two vertices

are 2-edge-connected if and only if they have the same label. The labels are defined by the bridge decomposition of the dominator trees and by the auxiliary components, as follows. Let $\widehat{G}_r$ be an auxiliary graph of G_s. We pick a *canonical vertex* for each SCC C of $\widehat{G}_r$, and denote by c_x the canonical vertex of the SCC that contains x. We define c_x^R for the SCC's of the auxiliary graphs of G_s^R analogously. We define the label of x as $label(x) = \langle r_x, c_x, r_x^R, c_x^R \rangle$.

▶ **Lemma 3.** *Let x and y be any vertices of G. Then, x and y are 2-edge-connected if and only if $label(x) = label(y)$.*

▶ **Theorem 4.** *The 2ECB labeling algorithm computes the 2-edge-connected blocks of a strongly connected digraph in linear time.*

Incremental dominators and incremental SCC's. We will use two other building blocks for our new algorithm, namely incremental algorithms for maintaining dominator trees and SCC's. As shown in [15], the dominator tree of a flow graph with n vertices can be maintained in $O(m \min\{n, k\} + kn)$ time during a sequence of k edge insertions, where m is the total number of edges after all insertions. For maintaining the SCC's of a digraph incrementally, Bender et al. [4] presented an algorithm that can handle the insertion of m edges in a digraph with n vertices in $O(m \min\{m^{1/2}, n^{2/3}\})$ time. Since we aim at an $O(mn)$ bound, we maintain the SCC's with a simpler data structure based on topological sorting [29], augmented so as to handle cycle contractions, as suggested by [17]. We refer to this data structure as IncSCC, and we will use it both for maintaing the SCC's of the input graph, and the auxiliary components (i.e., the SCC's of the auxiliary graphs). We maintain the SCC's and a topological order for them. Each SCC is represented by a canonical vertex, and the partition of the vertices into SCC's is maintained through a set union data structure [34, 36]. The data structure supports $unite(p, q)$, which, given canonical vertices p and q, merges the SCC's containing p and q into one new SCC and makes p the canonical vertex of the new SCC. It also supports $find(v)$, which returns the canonical vertex of the SCC containing v. Here we use the abbreviation $f(v)$ to stand for $find(v)$. The topological order is represented by a simple numbering scheme, where each canonical vertex is numbered with an integer in the range $[1, n]$, so that if (u, v) is an edge of G, then either $f(u) = f(v)$ (u and v are in the same SCC) or $f(u)$ is numbered less than $f(v)$ (when u and v are in different SCC's). With each canonical vertex p we store a list $out(p)$ of edges leaving vertices that are in the same SCC as p, i.e., edges (u, v) with $f(u) = p$. Note that $out(p)$ may contain multiple vertices in the same SCC (i.e., vertices u and v with $f(u) = f(v)$), due to the SCC contractions (and shortcut edges, in case of the auxiliary components) during edge insertions. Also, $out(p)$ may contain loops, that is, vertices v with $f(v) = p$. Each out list is stored as a doubly linked circular list, so that we can merge two lists and delete a vertex from a list in $O(1)$. When the incremental SCC data structure detects that a new SCC is formed, it locates the SCC's that are merged and chooses a canonical vertex for the new SCC. The IncSCC data structure can handle m edge insertions in a total of $O(mn)$ time.

3 Incremental 2-edge-connectivity in strongly connected digraphs

To maintain the 2-edge-connected blocks of a strongly connected digraph during edge insertions, we design an incremental version of the labeling algorithm of Section 2. In order to respond to the insertion of an edge, we have to update the vertex labels, so we need to update both the bridge decomposition of D and D^R, and the strongly connected components of the resulting auxiliary graphs. Note, in particular, that the second task involves moving

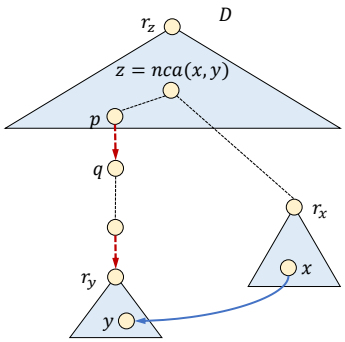

Figure 4 The bridge decomposition of D before the insertion of a new edge (x, y).

and merging vertices from one auxiliary graph to another. If labels are maintained explicitly, one can answer in $O(1)$ time queries on whether two vertices are 2-edge-connected, and report in $O(n)$ time all the 2-edge-connected blocks. Let (x, y) be the edge to be inserted. We say that vertex v is *affected* by the update if $d(v)$ (its parent in D) changes. Let $Dom(v)$ denote the set of all vertices that dominate v: note that $Dom(v)$ may change even if v is not affected. Similarly, an auxiliary component (resp., auxiliary graph) is affected if it contains an affected vertex.

We let $nca(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D. We also denote by $D[u, v]$ the path from u to v in D. If $nca(x, y)$ and y are in different subtrees in the bridge decomposition of D before the insertion of the edge (x, y), we let (p, q) be the first bridge encountered on the path $D[nca(x, y), y]$ (Figure 4). We denote by $depth(v)$ the depth of vertex v in D. Most of the proofs in this section will be given in the full paper.

Affected vertices and canceled bridges. There are affected vertices after the insertion of (x, y) if and only if $nca(x, y)$ is not a descendant of $d(y)$ [32]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [3].

▶ **Lemma 5.** ([15]) *A vertex v is affected after the insertion of edge (x, y) if and only if $depth(nca(x, y)) < depth(d(v))$ and there is a path π in G from y to v such that $depth(d(v)) < depth(w)$ for all $w \in \pi$. If v is affected, then it becomes a child of $nca(x, y)$ in D.*

The algorithm in [15] applies Lemma 5 to identify affected vertices by starting a search from y (if y is not affected, then no other vertex is). We assume that the outgoing and incoming edges of each vertex are maintained as linked lists, so that a new edge can be inserted in $O(1)$, and that the dominator tree D is represented by the parent function d. We also maintain the depth of vertices in D. We say that a vertex v is *scanned*, if the edges leaving v are examined during the search for affected vertices, and that it is *visited* if there is a scanned vertex u such that (u, v) is an edge in G. Every scanned vertex is either affected or a descendant of an affected vertex in D. By Lemma 5, a visited vertex v is scanned if $depth(nca(x, y)) < depth(d(v))$. Let (u, v) be a bridge of G_s. We say that (u, v) is *canceled* by the insertion of edge (x, y) if it is no longer a bridge after the insertion. We say that (u, v) is *locally canceled* if (u, v) is a canceled bridge and v is not affected. We need to treat the case of locally canceled bridges separately because in such an event the bridge decomposition of D changes, even if D remains the same. Note that if (u, v) is locally canceled, then $u = nca(x, y)$. In the next lemmata, we consider the effect of the insertion of edge (x, y) on

the bridges of G_s, and relate the affected and scanned vertices with the auxiliary components. Recall that (p, q) is the first bridge encountered on the path $D[nca(x, y), y]$ (Figure 4), $D(v)$ denotes the descendants of v in D, and $G[C]$ is the subgraph induced by the vertices in C.

▶ **Lemma 6.** *Suppose that bridge (p, q) is not locally canceled after the insertion of (x, y). Let $z = nca(x, y)$ and let v be an affected vertex such that $r_v \neq r_z$. All vertices reachable from v in $G[D(q)]$ are either affected or scanned.*

▶ **Lemma 7.** *Let $e = (u, v)$ be a bridge of G_s that is canceled by the insertion of edge (x, y). Then (i) y is a descendant of v in D, and (ii) y is in the same e-dominated component as v.*

▶ **Corollary 8.** *A bridge $e = (u, v)$ of G_s is canceled by the insertion of edge (x, y) if and only if $depth(nca(x, y)) \leq depth(u)$ and there is a path π in G from y to v such that $depth(u) < depth(w)$ for all $w \in \pi$.*

By Corollary 8, we can use the incremental algorithm of [15] to detect canceled bridges, without affecting the $O(mn)$ bound. Indeed, suppose $e = (u, v)$ is a canceled bridge. By Lemma 7, y is a descendant of v in D and in the same e-dominated component as v. Hence, v will be visited by the search from y. If a bridge (u, v) is locally canceled, there can be vertices in D_v that are not scanned, and that after the insertion will be located in D_u, without having their depth changed. This is a difficult case for our analysis: fortunately, the following lemma shows that this case can happen only $O(n)$ times overall.

▶ **Lemma 9.** *Suppose (u, v) is a bridge of G_s that is locally canceled by the insertion of edge (x, y). Then (u, v) is no longer a strong bridge in G after the insertion.*

Note that a canceled bridge that is not locally canceled may still appear as a bridge in G_s^R after the insertion of edge (x, y). The next lemmata allow us to identify the necessary changes in the auxiliary components of the affected subgraphs and $\widehat{G}_{r_z}$. All lemmata assume that bridge (p, q) is not locally canceled after the insertion of (x, y) and that $z = nca(x, y)$.

▶ **Lemma 10.** *Let C be an affected auxiliary component of an auxiliary graph $\widehat{G}_r$ with $r \neq r_z$. Then C consists of a set of affected siblings in D and possibly some of their affected or scanned descendants in D.*

An auxiliary component is *scanned* if it contains a scanned vertex. As with vertices, affected auxiliary components are also scanned (the converse is not necessarily true).

▶ **Lemma 11.** *Let C be a scanned auxiliary component of an auxiliary graph $\widehat{G}_r$ with $r \neq r_z$. Then all vertices in C are scanned.*

We say that a vertex v is *moved* if it is located in an auxiliary graph $\widehat{G}_r$ with $r \neq r_z$ before the insertion of (x, y), and in $\widehat{G}_{r_z}$ after the insertion. Lemmata 10 and 11 imply that if an auxiliary component C contains a moved vertex, then all vertices in the component are also moved. We call such an auxiliary component *moved*. Now we describe how to find the moved auxiliary components that need to be merged. Let H be the subgraph of G induced by the scanned vertices in $D(q)$. We refer to H as the *scanned subgraph*.

▶ **Lemma 12.** *Let ζ and ξ be two distinct roots in the bridge decomposition of D, such that $\zeta, \xi \neq r_z$, and D_ζ and D_ξ are contained in $D(q)$. Let C_ζ and C_ξ be scanned components in $\widehat{G}_\zeta$ and $\widehat{G}_\xi$, respectively. Then C_ζ and C_ξ are strongly connected in $G[D(q)]$ if and only if they are strongly connected in H.*

Now we introduce a dummy root r^* in H, together with an edge (v, r^*) for each scanned vertex v that has a leaving edge (v, w) such that $w \in D_z$ and w is in the auxiliary component of p in $\widehat{G}_{r_z}$. We denote this graph by H^*.

▶ **Lemma 13.** *A scanned vertex $v \notin D_z$ is strongly connected in $G[D(r_z)]$ to a vertex $w \in D_z$ if and only if r^* is reachable from v in H^*. In this case, v and p are also strongly connected in $G[D(r_z)]$.*

The Algorithm. We describe next our incremental algorithm for maintaining the 2-edge-connected blocks of a strongly connected digraph G. We refer to this algorithm as SCInc2ECB(G). We initialize the algorithm and the associated data structures by executing the labeling algorithm of Section 2. Algorithm Initialize(G, s), shown below, computes the dominator tree D, the set of bridges Br of flow graph G_s, the bridge decomposition $\mathcal{D}$ of D, and the corresponding auxiliary graphs $\widehat{G}_r$. Finally, for each auxiliary graph $\widehat{G}_r$, it finds its auxiliary components, computes the labels r_w and c_w for each vertex $w \in V_r$, and initializes an IncSCC data structure. The execution of Initialize(G^R, s) performs analogous steps in the reverse flow graph G_s^R.

Algorithm 1: Initialize(G, s)

```
1  Set s to be the designated start vertex of G.
2  Compute the dominator tree D and the set of bridges Br of the corresponding flow
     graph G_s.
3  Compute the bridge decomposition 𝒟 of D.
4  foreach root r in 𝒟 do
5  |  Compute the auxiliary graph Ĝ_r of r.
6  |  Compute the strongly connected components in Ĝ_r.
7  |  foreach strongly connected component C in Ĝ_r do
8  |  |  Choose a vertex v ∈ C as the canonical vertex of the auxiliary component C.
9  |  |  foreach vertex w ∈ C do
10 |  |  |  Set r_w = r and c_w = v.
11 |  |  end
12 |  end
13 |  Initialize a IncSCC data structure for Ĝ_r.
14 end
```

Algorithm 2: SCInsertEdge(G, e)

```
1  Let s be the designated start vertex of G, and let e = (x, y).
2  Compute the nearest common ancestor z and z^R of x and y in D and D^R respectively.
3  Update the dominator trees D and D^R, and return the lists S and S^R of the vertices
     that were scanned in D and D^R respectively.
4  if a bridge is locally canceled in G_s or in G_s^R then
5  |  Execute Initialize(G, s) and Initialize(G^R, s).
6  else
7  |  Execute UpdateAC(𝒟, z, x, y, S) and UpdateAC(𝒟^R, z^R, y, x, S^R).
8  end
```

Algorithm 3: UpdateAC$(\mathcal{D}, z, x, y, L)$

Let r_z be root of the tree D_z in $\mathcal{D}$ that contains z.
Let $c_{x'}$ be the canonical vertex of the nearest ancestor x' of x in D such that $x' \in D_z$.
Let (p, q) be the first bridge on the path $D[z, y]$, and let c_p be the canonical vertex of p.
Form the scanned graph H^* that contains the scanned vertices $S \setminus D_z$ and the edges among them.
Compute the strongly connected components $\mathcal{C}$ of $H^* \setminus r^*$ and order them topologically.
Compute the components $\mathcal{C}^*$ of $\mathcal{C}$ that reach r^* in H^*.
foreach *strongly connected component C in $\mathcal{C}^*$ that is moved* **do**
    Merge C with the component of c_p.
end
forall *strongly connected components in $\mathcal{C} \setminus \mathcal{C}^*$ that are moved* **do**
    Insert the components in the topological order of $\widehat{G}_{r_z}$ just after the component of c_p.
end
foreach *vertex $w \in S$* **do**
    if *w is moved to $\widehat{G}_{r_z}$* **then** set $r_w = r_z$.
end
Update the lists of out edges in the IncSCC data structures of $\widehat{G}_{r_z}$ and of the affected auxiliary graphs.
Insert edge $(c_{x'}, y)$ in the list of outgoing edges of $c_{x'}$ and update the IncSCC data structure of $\widehat{G}_{r_z}$.

When a new edge $e = (x, y)$ is inserted, algorithm SCInc2ECB executes procedure SCInsertEdge(G, e), which updates dominator trees D and D^R, together with the corresponding bridge decompositions. It also finds the set of scanned vertices in G_s and G_s^R. If a bridge of D or D^R is locally cancelled, then we restart the algorithm by executing Initialize. Otherwise, we need to update the auxiliary components in G_s and G_s^R. These updates are handled by procedure UpdateAC. Before describing UpdateAC, we provide some details on the implementation of the IncSCC data structures, which maintain the auxiliary components of each auxiliary graph $\widehat{G}_r$ using the "one-way search" structure of [17, Sections 2 and 6]. Since we need to insert and delete canonical vertices, we augment this data structure as follows. We maintain the canonical vertices of each auxiliary component in a linked list L_r, arranged according to the given topological order of $\widehat{G}_r$. For each vertex v in L_r, we also maintain a rank in L_r which is an integer in $[1, n]$ such that for any two canonical vertices u and v in L_r, $rank(u) < rank(v)$ if and only if u precedes v in L_r. The ranks of all vertices can be stored in a single array of size n. Also, with each canonical vertex w, we store a pointer to the location of w in L. We represent L_r with a doubly linked list so that we can insert and delete a canonical vertex in constant time. When we remove vertices from a list L_r we do not need to update the ranks of the remaining vertices in L_r. The insertion of an edge (x, y) may remove vertices from various lists L_r, but may insert vertices only in L_{r_z}. After these insertions, we recompute the ranks of all vertices in L_{r_z} just by traversing the list and assigning rank i to the i-th vertex in the list. We maintain links between an original edge e, stored in the adjacency lists of G, and at most one copy of e in a *out* list of IncSCC. This enables us to keep for each shortcut edge $e' = (v', w)$ a one-to-one correspondence with the original edge $e = (v, w)$ that created e'. We do that because if an ancestor of v is moved

to the auxiliary graph $\widehat{G}_{r_z}$ that contains v' ($v' = p$ in Figure 5), then e may correspond to a different shortcut edge or it may even become an ordinary edge of $\widehat{G}_{r_z}$. Using this mapping we can update the *out* lists of IncSCC. To initialize the IncSCC structure of an auxiliary graph, we compute a topological order of the auxiliary components in $\widehat{G}_r$, and create the list of outgoing edges $out(v)$ for each canonical vertex v.

If inserting edge (x, y) does not locally cancel a bridge in G_s and G_s^R, then we update the auxiliary components of G_s using procedure UpdateAC($\mathcal{D}, z, x, y, S$), where $\mathcal{D}$ is the updated bridge decomposition of D, $z = nca(x, y)$, and S is a list of the vertices scanned during the update of D. We do the same to update the auxiliary components of G_s^R. Procedure UpdateAC first computes the auxiliary components that are moved to $\widehat{G}_{r_z}$, possibly merging some of them, and then inserts the edge (x, y) as an original or a shortcut edge of $\widehat{G}_{r_z}$, depending on whether $x \in D_{r_z}$ or not. Note that the insertion of (x, y) may cause the creation of a new auxiliary component in $\widehat{G}_{r_z}$. Now we specify some further details in the implementation of UpdateAC. The vertices that are moved to $\widehat{G}_{r_z}$ are the scanned vertices in S that are not descendants of a strong bridge. Hence, we can mark the vertices that are moved to $\widehat{G}_{r_z}$ during the search for affected vertices. The next task is to update the *out* lists of the canonical vertices in $\widehat{G}_{r_z}$ and the affected auxiliary graphs. We process the list of scanned vertices S as follows. Let v be such a vertex. If v is not marked, i.e., is not moved to $\widehat{G}_{r_z}$, then we process the edges leaving v; otherwise, we process both the edges leaving v and the edges entering v. Suppose v is marked. Let (v, w) be an edge leaving v in G. If w is also in $\widehat{G}_{r_z}$ after the insertion, then we add the edge (v, w) in $out(f(v))$. Moreover, if w is not in S, then it was already located in $\widehat{G}_{r_z}$ before the insertion, so we delete the shortcut edge stored in $out(f(p))$. If w is not in $\widehat{G}_{r_z}$ after the insertion, then (v, w) is a bridge in D and we do nothing. Now consider an edge (w, v) entering v in G. If w is scanned, then we will process (w, v) while processing the edges leaving w. Otherwise, w remains a descendant of p, so we insert the edge (w, v) in $out(f(p))$. Now we consider the unmarked scanned vertices v. Let (v, w) an edge leaving v in G. If $w \in D_z$, we insert the edge (v, w) into $out(f(v'))$, where v' is the nearest marked ancestor of v in D. Otherwise, if $w \notin D(r_z)$, the edge (v'', w), where v'' is the nearest ancestor of v in D_w, already exists since v was a descendant of v'' before the insertion of (x, y). Next, we consider the updates in the L_r lists and the vertex ranks. While we process S, if we encounter a moved canonical vertex $v \in S$ that was located in an auxiliary graph $\widehat{G}_r$ with $r_z \neq r$, then we delete v from L_r. Note that we do not need to update the ranks of the remaining vertices in lists L_r with $r \neq r_z$. To update L_{r_z}, we insert the moved canonical vertices of the SCC's in $\mathcal{C} \setminus \mathcal{C}^*$, in a topological order of $H = H^* \setminus r^*$, just after $f(p)$. Then we traverse L_{r_z} and update the ranks of the canonical vertices. The final step is to actually insert edge (x, y) in the IncSCC data structure of $\widehat{G}_{r_z}$. We do that by adding (x, y) in $out(f(x'))$, where x' is the nearest ancestor of x in D_z. If $rank(f(x')) > rank(f(y))$, then we execute the forward-search procedure of IncSCC.

The proof of correctness of Algorithm SCInc2ECB will be given in the full paper.

Running time of SCInc2ECB. We analyze the running time of Algorithm SCInc2ECB. Recall that G is a strongly connected digraph with n vertices that undergoes a sequence of edge insertions. We let m be the total number of edges in G after all insertions ($m \geq n$). First, we bound the time spent by Initialize. This procedure is called twice in the beginning of the SCInc2ECB, and twice after each time a bridge in G_s or in G_s^R is locally canceled. Then, Lemma 9 implies that such an event can happen at most $2(n-1)$ times. Hence, there are at most $4n$ calls to Initialize, and since each execution takes $O(m)$ time, the total time spent on Initialize is $O(mn)$. Similarly, the dominator trees of G_s and G_s^R can be updated

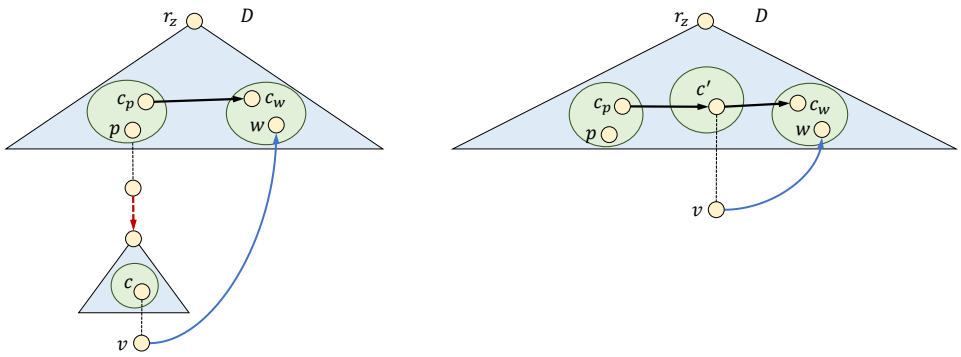

Figure 5 Before the insertion of (x, y), edge (v, w) corresponds to the shortcut edge (p, w) of $\widehat{G}_{r_z}$, and is stored in $out(c_p)$. An auxiliary component with canonical vertex c is affected by the insertion of (x, y) and is merged into a component with canonical vertex c' ($c' = c$ if the component is moved without merging with another component). Now c' becomes the canonical vertex of the nearest ancestor of v in D_z, and edge (v, w) is stored as a shortcut edge in $out(c')$.

in total $O(mn)$ time [15]. We next bound the total time required to update the auxiliary components. Consider an execution of UpdateAC. Let ν and μ, respectively, be the number of scanned vertices, after the insertion of edge (x, y), and their adjacent edges. The time to compute the affected subgraph H^*, compute the SCC's of $H^* \setminus r^*$, and the vertices that reach r^* is $O(\nu + \mu)$. In the same time, we can update the auxiliary components of $\widehat{G}_{r_z}$ and of the affected auxiliary graphs, their corresponding topological orders, and the *out* lists of the corresponding IncSCC data structures. Since each scanned vertex w is a descendant of an affected vertex, the depth of w decreases by at least one. Hence, the total time spent by UpdateAC for all insertions, excluding the execution of line 17, is $O(mn)$. It remains to bound the time required by the IncSCC data structures to handle the edge insertions in line 17 of UpdateAC. To do this, we extend the analysis from [17]. Note that we cannot immediately apply the analysis in [17], since here we have the complication that vertices and edges can be inserted to and removed from the IncSCC structures. We say that a vertex v and an edge e are *related* if there is a path that contains both v and e (in any order). Then, there are $O(mn)$ pairs of vertices and edges that can be related in all IncSCC structures for every auxiliary graph. We argue that each time the IncSCC structure traverses an edge (after the insertion in line 17 of UpdateAC), the cost of this action can be charged to a newly-related vertex-edge pair. Consider a vertex v and an edge $e = (u, w)$. Call the pair $\langle v, e\rangle$ *active* if v and e are in the same auxiliary graph $\widehat{G}_r$, and *inactive* otherwise. Note that since we identify shortcut edges with their corresponding original edge, e may actually appear in $\widehat{G}_r$ as an edge (u', w), where u' is the nearest ancestor of u in D_r. This fact, however, does not affect our analysis.

▶ **Lemma 14.** *The total number of edge traversals made during the forward searches in all IncSCC data structures is $O(mn)$.*

Proof. To prove the bound, it suffices to show that in all IncSCC data structures the total number of unrelated $\langle v, e\rangle$ pairs that are ever created is $O(mn)$. Consider an active pair $\langle v, e\rangle$ that becomes related in $\widehat{G}_r$. Then there is some path π in $G[D(r)]$ that contains both v and e. Suppose that the pair $\langle v, e\rangle$ later becomes active but unrelated in an auxiliary graph $\widehat{G}_{r'}$, where r' may be vertex r. Then π does not exist in $G[D(r')]$, which implies that some vertices of π are not descendants of r'. Then, by Lemma 1, π must contain the

bridge $(d(r'), r')$. Since π exists in $G[D(r)]$, the bridge $(d(r'), r')$ was a descendant of r before some insertion, and then became an ancestor of v. But this is impossible, since after an edge insertion, the new parent $d'(v)$ of v is on the path $D[s, d(v)]$. Hence, once a $\langle v, e\rangle$ pair becomes related, it can never become unrelated. The bound follows. ◀

▶ **Lemma 15.** *The total time to update all the IncSCC data structures is $O(mn)$.*

Proof. Updating the lists of out edges in the IncSCC data structures, and inserting or deleting canonical vertices can be charged to the cost of updating the dominator tree, and is thus $O(mn)$. By Lemma 14, all edge insertions that do not trigger merges of auxiliary components can be handled in $O(mn)$ time. The number of edge insertions that trigger merges of auxiliary components is at most $n - 1$, and each such insertion can be handled in $O(m + n)$ time, excluding unite operations. Taking into account also the total time for all unite operations yields the lemma. ◀

▶ **Theorem 16.** *The total running time of Algorithm SCInc2ECB for a sequence of edge insertions in a strongly connected digraph with n vertices is $O(mn)$, where m is the total number of edges in G after all insertions.*

Extension to general digraphs. Our approach can be extended to general (not strongly connected) digraphs, as shown in the following theorem. Details will be given in the full paper.

▶ **Theorem 17.** *We can maintain the 2-edge-connected blocks of a digraph with n vertices through a sequence of edge insertions in $O(mn)$ time, where m is the total number of edges in G after all insertions.*

References

1. A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th IEEE Symp. on Foundations of Computer Science, FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.
2. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
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. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, December 2015. doi:10.1145/2756553.
5. 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.
6. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of ACM*, 51(6):968–992, 2004. doi:10.1145/1039488.1039492.
7. C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008. doi:10.1007/s00453-007-9051-4.
8. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *Journal of ACM*, 44(5):669–696, September 1997. doi:10.1145/265910.265914.

9 W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:10.1016/j.jda.2013.10.003.

10 P. G. Franciosa, G. Gambosi, and U. Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information Processing Letters*, 61(2):113–120, 1997. doi:10.1016/S0020-0190(96)00202-5.

11 G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM Journal on Computing*, 14:781–798, 1985.

12 H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016. doi:10.1145/2764909.

13 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms*, pages 1988–2005, 2015.

14 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015.

15 L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symp. on Algorithms*, pages 491–502, 2012.

16 L. Georgiadis, G. F. Italiano, and N. Parotsidis. A New Framework for Strong Connectivity and 2-Connectivity in Directed Graphs. *ArXiv e-prints*, abs/1511.02913, November 2015. arXiv:1511.02913.

17 B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms*, 8(1):3:1–3:33, January 2012. doi:10.1145/2071379.2071382.

18 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015)*, 2015.

19 M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symp. on Foundations of Computer Science*, pages 664–672, 1995. doi:10.1109/SFCS.1995.492668.

20 M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of ACM*, 46(4):502–536, 1999.

21 M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM Journal on Computing*, 31(2):364–374, February 2002. doi:10.1137/S0097539797327209.

22 J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of ACM*, 48(4):723–760, July 2001. doi:10.1145/502090.502095.

23 G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(3):273–281, 1986. doi:10.1016/0304-3975(86)90098-8.

24 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.

25 R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015. doi:10.1051/ita/2015001.

26 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.

27 V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symp. on Foundations of Computer Science, FOCS'99*, pages 81–91, 1999. doi:10.1109/SFFCS.1999.814580.

28 S. Makino. An algorithm for finding all the k-components of a digraph. *International Journal of Computer Mathematics*, 24(3–4):213–221, 1988.

29 A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996. doi:10.1016/0020-0190(96)00075-0.

30 H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76–A.4:513–517, 1993.

31 M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proc. 48th IEEE Symp. on Foundations of Computer Science*, FOCS'07, pages 263–271, 2007. doi:10.1109/FOCS.2007.54.

32 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 Symp. on Principles of programming languages*, pages 287–296, 1994.

33 R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

34 R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.

35 R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

36 R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31(2):245–81, 1984.

37 M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proc. 32nd ACM Symp. on Theory of Computing*, STOC'00, pages 343–350, 2000. doi:10.1145/335305.335345.

38 M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Algorithm Theory – SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory,*, pages 384–396, 2004. doi:10.1007/978-3-540-27810-8_33.