2-Edge Connectivity in Directed Graphs

LOUKAS GEORGIADIS, University of Ioannina
GIUSEPPE F. ITALIANO, Università di Roma "Tor Vergata"
LUIGI LAURA, "Sapienza" Università di Roma
NIKOS PAROTSIDIS, Università di Roma "Tor Vergata"

Edge and vertex connectivity are fundamental concepts in graph theory. While they have been thoroughly studied in the case of undirected graphs, surprisingly, not much has been investigated for directed graphs. In this article, we study 2-edge connectivity problems in directed graphs and, in particular, we consider the computation of the following natural relation: We say that two vertices v and w are 2-*edge-connected* if there are two edge-disjoint paths from v to w and two edge-disjoint paths from w to v. This relation partitions the vertices into blocks such that all vertices in the same block are 2-edge-connected. Differently from the undirected case, those blocks do not correspond to the 2-edge-connected components of the graph. The main result of this article is an algorithm for computing the 2-edge-connected blocks of a directed graph in linear time. Besides being asymptotically optimal, our algorithm improves significantly over previous bounds. Once the 2-edge-connected blocks are available, we can test in constant time if two vertices are 2-edge-connected. Additionally, when two query vertices 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. We are also able to compute in linear time a sparse certificate for this relation, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph, where n is the number of vertices.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Graphs and networks; Lists, stacks, and queues; Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms; Trees*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Connectivity, directed graph, dominators, flow graph

ACM Reference Format:
Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2016. 2-edge connectivity in directed graphs. ACM Trans. Algorithms 13, 1, Article 9 (October 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/2968448

A preliminary version of this work appeared in *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms,* pp. 1988–2005, 2015. Research at Università di Roma "Tor Vergata" partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).
Authors' addresses: L. Georgiadis, Department of Computer Science and Engineering, University of Ioannina, GR 45110 - Ioannina, Greece; email: loukas@cs.uoi.gr; G. F. Italiano and N. Parotsidis, Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata", Roma, Italy; email: giuseppe.italiano@uniroma2.it, nikos.parotsidis@uniroma2.it; L. Laura, Dipartimento di Ingegneria Informatica, Automatica e Gestionale e Centro di Ricerca per il Trasporto e la Logistica (CTL), "Sapienza" Università di Roma, Roma, Italy; email: laura@dis.uniroma1.it.

DOI: http://dx.doi.org/10.1145/2968448

1. INTRODUCTION

Let $G = (V, E)$ be an *undirected* (resp., *directed*) graph, with m edges and n vertices. Throughout the article, we use, interchangeably, the term *directed graph* and digraph. Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications [Bang-Jensen and Gutin 2002; Nagamochi and Ibaraki 2008]. As an example, we mention the computation of disjoint paths in routing and reliable communication, both in undirected and directed graphs [Guo et al. 2003; Itai and Rodeh 1988].

Throughout the article, we assume that the reader is familiar with the standard graph terminology, as contained, for instance, in Cormen et al. [1991]. An *undirected path* (resp., *directed path*) in G is a sequence of vertices $v_1, v_2, \dots, v_k$, such that edge $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \dots, k-1$. An undirected graph G is *connected* if there is an undirected path from each vertex to every other vertex. The *connected components* of an undirected graph are its maximal connected subgraphs. A directed graph G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* of a directed graph are its maximal connected subgraphs.

Given an undirected graph $G = (V, E)$, an edge is a *bridge* if its removal increases the number of connected components of G. Graph G is 2-edge-connected if it has no bridges. The 2-*edge-connected components* of G are its maximal 2-edge-connected subgraphs. Two vertices v and w are 2-edge-connected if there are two edge-disjoint paths between v and w: we denote this relation by $v \leftrightarrow_{2\mathrm{e}} w$. Equivalently, by Menger's Theorem [Menger 1927], v and w are 2-edge-connected if the removal of any edge leaves them in the same connected component. Analogous definitions can be given for 2-vertex connectivity. In particular, a vertex is an *articulation point* if its removal increases the number of connected components of G. A graph G is 2-vertex-connected if it has at least three vertices and no articulation points. The 2-*vertex-connected components* of G are its maximal 2-vertex-connected subgraphs. Note that the condition on the minimum number of vertices in a 2-vertex-connected graph disallows degenerate 2-vertex-connected components consisting of one single edge. Two vertices v and w are 2-vertex-connected if there are two internally vertex-disjoint paths between v and w: we denote this relation by $v \leftrightarrow_{2\mathrm{v}} w$. If v and w are 2-vertex-connected, then Menger's Theorem implies that the removal of any vertex different from v and w leaves them in the same connected component. The converse does not necessarily hold, since v and w may be adjacent but not 2-vertex-connected. It is easy to show that $v \leftrightarrow_{2\mathrm{e}} w$ (resp., $v \leftrightarrow_{2\mathrm{v}} w$) if and only if v and w are in the same 2-edge-connected (resp., 2-vertex-connected) component. All bridges, articulation points, 2-edge-, and 2-vertex-connected components of undirected graphs can be computed in linear time essentially by the same algorithm [Tarjan 1972].

The notions of 2-edge and 2-vertex connectivity were naturally extended to directed graphs in Italiano et al. [2012]. Given a digraph G, an edge (resp., a vertex) is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of strongly connected components of G. A digraph G is 2-edge-connected if it has no strong bridges; G is 2-vertex-connected if it has at least three vertices and no strong articulation points. The 2-edge-connected (resp., 2-vertex-connected) components of G are its maximal 2-edge-connected (resp., 2-vertex-connected) subgraphs. Again, the condition on the minimum number of vertices disallows for degenerate 2-vertex-connected components consisting of two mutually adjacent vertices (i.e., two vertices v and w and the two edges (v, w) and (w, v)).

Similarly to the undirected case, we say that two vertices v and w are 2-edge-connected (resp., 2-vertex-connected), and we denote this relation by $v \leftrightarrow_{2\mathrm{e}} w$ (resp., $v \leftrightarrow_{2\mathrm{v}} w$), if there are two edge-disjoint (resp., internally vertex-disjoint) directed paths from v to w and two edge-disjoint (resp., internally vertex-disjoint) directed paths from

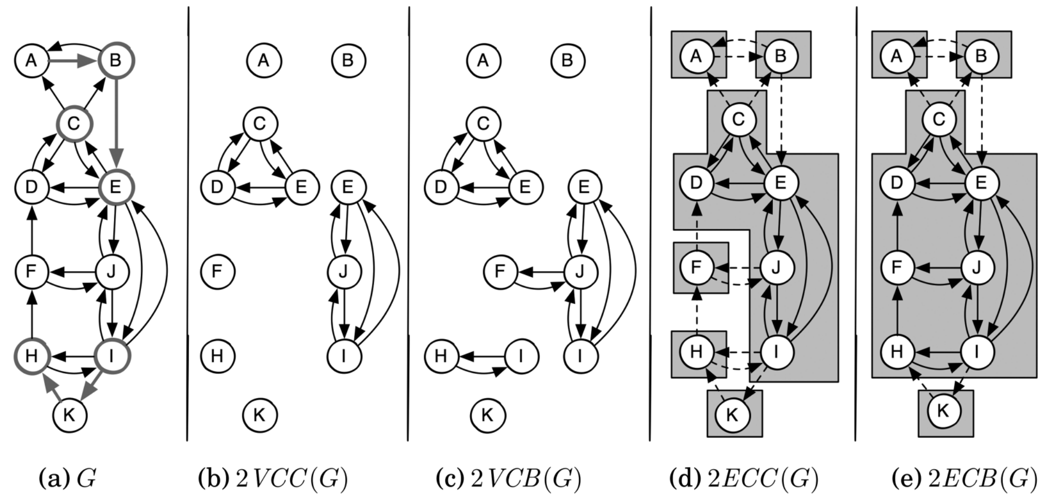

Fig. 1. (a) A strongly connected digraph G, with strong articulation points and strong bridges shown in red (better viewed in color). (b) The 2-vertex-connected components of G. (c) The 2-vertex-connected blocks of G. (d) The 2-edge-connected components of G. (e) The 2-edge-connected blocks of G.

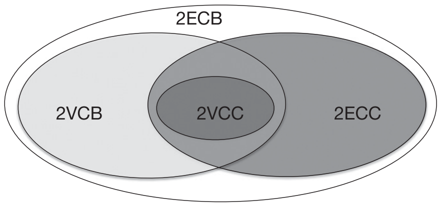

Fig. 2. The relation among various notions of 2-connectivity in directed graphs. Two vertices that are 2-edge-connected (resp., 2-vertex-connected) are in the same 2-edge-connected (resp., 2-vertex-connected) block but not necessarily in the same 2-edge-connected (resp., 2-vertex-connected) component. Also, a 2-vertex-connected component is included in a 2-edge-connected component.

w to v. (Note that a path from v to w and a path from w to v need not be edge-disjoint or vertex-disjoint). It is easy to see that $v \leftrightarrow_{2e} w$ if and only if the removal of any edge leaves v and w in the same strongly connected component. Similarly, $v \leftrightarrow_{2v} w$ implies that the removal of any vertex different from v and w leaves v and w in the same strongly connected component. We define a 2-*edge-connected block* (resp., 2-*vertex-connected block*) of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ (resp., $u \leftrightarrow_{2v} v$) for all $u, v \in B$. It can be easily seen that, differently from undirected graphs, in digraphs, 2-edge- and 2-vertex-connected blocks do not correspond to 2-edge- and 2-vertex-connected components, as illustrated in Figure 1. Two vertices may be 2-edge-connected (resp., 2-vertex-connected) but lie in different 2-edge-connected (resp., 2-vertex-connected) components. Furthermore, these notions seem to have a much richer and more complicated structure in digraphs, as depicted in Figure 2. Just to give an example, we observe that while in the case of undirected connected graphs the 2-edge-connected components (which correspond to the 2-edge-connected blocks) are exactly the connected components left after the removal of all bridges, for directed, strongly connected graphs, the 2-edge-connected components, the 2-edge-connected blocks, and the strongly connected components left after the removal of all strong bridges are not necessarily the same; See Figure 3.

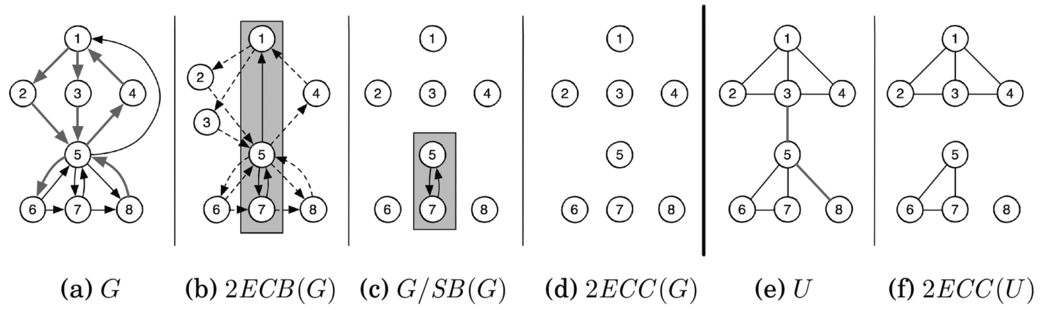

Fig. 3. (a) A digraph G with strong bridges shown in red; (b) The 2-edge-connected blocks of G; (c) The strongly connected components left after removing all the strong bridges from G; (d) The 2-edge-connected components of G. (e) An undirected graph U with bridges shown in red; (f) The 2-edge-connected components of U, corresponding to the 2-edge-connected blocks and to the connected components left after the removal of all bridges of U.

It is, thus, not surprising that, despite being complete analogs of the corresponding notions on undirected graphs, 2-edge- and 2-vertex connectivity problems appear to be more difficult on digraphs. In particular, although all the strong bridges and strong articulation points of a digraph can be found in linear time [Italiano et al. 2012], computing efficiently, say in linear time, the 2-edge- and 2-vertex-connected components, or the 2-edge- and 2-vertex-connected blocks in a digraph has been an elusive goal. A simple algorithm for computing the 2-edge-connected components can be obtained by repeatedly removing all the strong bridges in the graph (and repeating this process until no strong bridges are left). Since at each round all the strong bridges can be computed in $O(m + n)$ time [Italiano et al. 2012] and there can be at most $O(n)$ rounds, the total time taken by this algorithm is $O(mn)$. The same bound was previously achieved by Nagamochi and Watanabe [1993]. As for 2-vertex connectivity, Erusalimskii and Svetlov [1980] proposed an algorithm that reduces the problem of computing the 2-vertex-connected components of a digraph to the computation of the 2-vertex-connected components in an undirected graph, but did not analyze the running time of their algorithm. Jaberi [2016] showed that the algorithm of Erusalimskii and Svetlov has $O(nm^2)$ running time, and proposed two different algorithms with running time $O(mn)$. Both algorithms follow, substantially, the same high-level approach as the simple algorithm for computing the 2-edge-connected components of a digraph. Very recently, Henzinger et al. [2015] improved the $O(mn)$-time bound for computing the 2-edge and the 2-vertex-connected components to $O(n^2)$ by using a hierarchical graph decomposition technique.

A simple algorithm for computing the 2-edge- or 2-vertex-connected blocks of a digraph takes $O(mn)$ time: Given a vertex v, one can find in linear time all the vertices that are 2-edge- or 2-vertex-connected with v with the help of dominator trees. Since, in the worst case, this step must be repeated for all vertices v, the total time required by the simple algorithm is $O(mn)$. Very recently, and independently of our work, Jaberi [2015] presented algorithms for computing the 2-vertex-connected and 2-edge-connected blocks. His algorithms require $O(n \cdot \min\{m, b^*n\})$ time for computing the 2-edge-connected blocks and $O(n \cdot \min\{m, (a^* + b^*)n\})$ time for computing the 2-vertex-connected blocks, where a^* and b^* are, respectively, the number of strong articulation points and strong bridges in the digraph G. Since both a^* and b^* can be as large as $O(n)$, both bounds are $O(mn)$ in the worst case.

From the above discussion, it is clear that, differently from the case of undirected graphs, for digraphs, there is a huge gap between the $O(m + n)$ time bound for computing all connectivity cuts (strong bridges and strong articulation points), and the $O(n^2)$ and $O(mn)$ time bounds for computing the connectivity components and blocks, respectively (2-edge- and 2-vertex-connected components and 2-edge- and 2-vertex-connected blocks). Thus, it seems quite natural to ask whether there is a natural barrier for those problems or whether they could be solved faster in linear time.

In this article, we answer this question by presenting the first linear-time algorithm to compute the 2-edge-connected blocks of a digraph. Our algorithm is not only asymptotically optimal, but it also improves significantly over previous bounds. Furthermore, the ability to compute the 2-edge-connected blocks of a digraph in linear time seems a significant step, especially as it is the first real progress on this extremely natural problem, starting from the foundational work done 40 years ago for undirected graphs.

Our approach hinges on two different algorithms. The first is a simple iterative algorithm that builds the 2-edge-connected blocks by removing one strong bridge at a time. The second algorithm is more involved and recursive: The main idea is to consider simultaneously how different strong bridges partition vertices with the help of dominator trees. Although both algorithms run in $O(mn)$ time in the worst case, we show that a sophisticated combination of the iterative and the recursive method is able to achieve the claimed linear-time bound. Using our algorithm for 2-edge-connected blocks, we can preprocess a digraph in linear time and then answer in constant time queries on whether any two vertices are 2-edge-connected. Additionally, when two query vertices 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. We also show how to compute in linear time a sparse certificate for 2-edge-connected blocks, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph.

2. FLOW GRAPHS, DOMINATORS, AND BRIDGES

In this section, we introduce some terminology that will be useful throughout the article. A *flow graph* is a digraph such that every vertex is reachable from a distinguished start vertex. Let $G = (V, E)$ be the input digraph, which we assume to be strongly connected. (If not, we simply treat each strongly connected component separately.) For any vertex $s \in V$, we denote by $G(s) = (V, E, s)$ the corresponding flow graph with start vertex s; all vertices in V are reachable from s since G is strongly connected. The *dominator relation* in $G(s)$ is defined as follows: A vertex u is a *dominator* of a vertex w (*u dominates w*) if every path from s to w contains u; u is a *proper dominator* of w if u dominates w and $u \neq w$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* $D(s)$: u dominates w if and only if u is an ancestor of w in $D(s)$. If $w \neq s$, $d(w)$, the parent of w in $D(s)$, is the *immediate dominator* of w: It is the unique proper dominator of w that is dominated by all proper dominators of w. An edge $(u, w) \in E$ is a *bridge* in $G(s)$ if all paths from s to w in G include (u, w).[1]

Lengauer and Tarjan [1979] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [Tarjan 1975]. Subsequently, several linear-time algorithms were discovered [Alstrup et al. 1999; Buchsbaum et al. 2008, 1998; Fraczak et al. 2013; Gabow 2016; Georgiadis and Tarjan 2004]. Tarjan [1974a] showed that the bridges of flow graph $G(s)$ can be computed in $O(m)$ time given $D(s)$. He also presented an $O(m\alpha(m, n))$-time algorithm to compute bridges that uses static tree set union to contract strongly connected subgraphs in G [Tarjan 1976]. The Gabow-Tarjan static tree disjoint set union algorithm [Gabow and Tarjan 1985] reduces the running time of

[1]Throughout the article, to avoid danger of ambiguity, we use consistently the term *bridge* to refer to a bridge of a flow graph $G(s)$ and the term *strong bridge* to refer to a strong bridge in the original graph G.

this algorithm to $O(m)$ on a RAM. Buchsbaum et al. [2008] gave an $O(m)$-time pointer-machine algorithm.

Italiano et al. [2012] showed that the strong articulation points of G can be computed from the dominator trees of $G(s)$ and $G^R(s)$, where s is an arbitrary start vertex and G^R is the the digraph that results from G after reversing edge directions; similarly, the strong bridges of G correspond to the bridges of $G(s)$ and $G^R(s)$. This gives the following bound on the number of strong bridges.

LEMMA 2.1. *Any digraph with n vertices has at most* $2n - 2$ *strong bridges.*

Experimental studies for algorithms that compute dominators, strong bridges, and strong articulation points are presented in Firmani et al. [2012] and Georgiadis et al. [2014]. The experimental results show that the corresponding fast algorithms given in Fraczak et al. [2013], Italiano et al. [2012], Lengauer and Tarjan [1979], and Tarjan [1976] perform very well in practice, even on very large graphs.

3. COMPUTING THE 2-EDGE-CONNECTED BLOCKS

We recall that $u \leftrightarrow_{2e} w$ denotes that vertices u and w are 2-edge-connected, and that a *2-edge-connected block* of a digraph $G = (V, E)$ is a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} w$ for all $u, w \in B$.

THEOREM 3.1. *The* 2*-edge-connected blocks of a digraph* $G = (V, E)$ *form a partition of* V.

PROOF. We show that $\leftrightarrow_{2e}$ is an equivalence relation. The relation is by definition reflexive and symmetric, so it remains to show that it is also transitive when G has at least three vertices. Let u, v, and w be three distinct vertices such that $u \leftrightarrow_{2e} v$ and $v \leftrightarrow_{2e} w$. Consider any u-w cut (U, W), where $u \in U$ and $w \in W$. Let k' be the number of edges directed from U to W. We will show that $k' \geq 2$. If $v \in U$, then $v \leftrightarrow_{2e} w$ implies that $k' \geq 2$. Otherwise, $v \in W$, and $u \leftrightarrow_{2e} v$ implies that $k' \geq 2$. A completely analogous argument applies to the edges directed from W to U. The fact that $u \leftrightarrow_{2e} w$ now follows from Menger's Theorem [Menger 1927]. □

Throughout, we use the notation $[v]_{2e}$ to denote the 2-edge-connected block containing vertex $v \in V$. We can generalize the 2-edge-connected relation for $k \geq 2$ edge-disjoint paths: The proof of Theorem 3.1 can be extended to show that this relation also defines a partition of V into k-edge-connected blocks. By Theorem 3.1, once the 2-edge-connected blocks are available, it is easy to test in constant time if two vertices are 2-edge-connected.

Next, we develop algorithms that compute the 2-edge-connected blocks of a digraph G. Clearly, we can assume without loss of generality that G is strongly connected, so $m \geq n$. If not, then we process each strongly connected component separately; if $u \leftrightarrow_{2e} v$, then u and v are in the same strongly connected component S of G, and moreover, any vertex on a path from u to v or from v to u also belongs in S. We begin with a simple algorithm that removes a single strong bridge at a time. In order to get a more efficient solution, we need to consider simultaneously how different strong bridges partition the vertex set. We present a recursive algorithm that does this with the help of dominator trees. Although both these algorithms run in $O(mn)$ time in the worst case, we finally show that a careful combination of them is able to achieve linear time.

3.1. A Simple Algorithm

Let u and v be two distinct vertices in G. We say that a strong bridge *e separates u from v* if all paths from u to v contain edge e. In this case, u and v must belong to different strongly connected components of $G \backslash e$. This simple observation gives a characterization

```
Algorithm Simple2ECB: Computation of the 2-edge-connected blocks of a
strongly connected digraph $G = (V, E)$
Step 1: Initialize the 2-edge-connected blocks as $[v]_{2e} = V$. (Start from the trivial partition containing only one block.)
Step 2: Compute the strong bridges of $G$.
Step 3: For each strong bridge $e$ do:
  Step 3.1: Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus e$.
  Step 3.2: Let $\{[v_1]_{2e}, \ldots, [v_l]_{2e}\}$ be the current 2-edge-connected blocks. Refine the
      partition into blocks by computing the intersections $[v_i]_{2e} \cap S_j$ for all $i = 1, \ldots, l$
      and $j = 1, \ldots, k$.
```

Fig. 4. Algorithm Simple2ECB.

of the 2-edge-connected blocks in terms of the strong bridges. In particular, one can obtain the 2-edge-connected blocks of G by simply computing the strongly connected components of $G \setminus e$ for every strong bridge e, as illustrated by Algorithm Simple2ECB in Figure 4.

LEMMA 3.2. *Algorithm Simple2ECB runs in $O(mb^*)$ time, where b^* is the number of strong bridges of G.*

PROOF. The strong bridges of G can be computed in linear time by Italiano et al. [2012]. In each iteration of Step 3, we can compute the strongly connected components of $G \setminus e$ in linear time [Tarjan 1972]. As we discover the i-th strongly connected component, we assign label i ($i \in \{1, \ldots, n\}$) to the vertices in S_i. Then, the refinement of the current blocks in Step 3.1 can be done in $O(n)$ time with bucket sorting as follows. We initialize an array Bucket of size n, where the i-th entry Bucket[i] is a bucket associated with the strongly connected component S_i. Each bucket is implemented as a singly linked list, initially empty. We keep the indices i of the nonempty buckets Bucket[i] in a stack Index, so that we can re-initialize Bucket fast after processing each block. We refine each block of the current partition one at a time. To refine block B, we insert each vertex of B in the appropriate bucket; that is, a vertex of B with label i is inserted into the bucket Bucket[i]. After inserting all vertices of B into their corresponding buckets, we pop from Index the indices of the non-empty buckets. For each such index i, the vertices in Bucket[i] form a new block of the partition. So each iteration can be implemented in $O(m + n) = O(m)$ time. Since there are exactly b^* iterations, the lemma follows. □

Note that the above bound is $O(mn)$ in the worst case, since for any digraph $b^* \leq 2n-2$ by Lemma 2.1. We remark that deleting all strong bridges (at once) will not produce a correct result, as it can be easily seen from Figures 3(b) and 3(c). Despite the fact that removing a single strong bridge at a time does not yield an efficient algorithm, we will make use of this idea, in a more restricted way, in the linear-time algorithm described in Section 3.4.

3.2. Dominator Tree Partitions

In our effort to obtain a faster algorithm, we investigate how multiple strong bridges affect the partition of the vertices into blocks. We do this by selecting an arbitrary start vertex s and by using the dominator tree $D(s)$ of the flow graph $G(s)$, as follows. First, we consider the computation of the 2-edge-connected block that contains a specific vertex v. Let w be a vertex other than v. We say that w is 2-*edge-connected from* v if there are two edge disjoint paths from v to w. Analogously, we say that w is 2-*edge-connected to*

v if there are two edge disjoint paths from w to v. We divide the computation of $[v]_{2e}$ in two parts, where the first part finds the set of vertices $[v]_{\overrightarrow{2e}}$ that are 2-edge-connected from v, and the second part finds the set $[v]_{\overleftarrow{2e}}$ of vertices that are 2-edge-connected to v. Then, $[v]_{2e}$ is formed by the intersection of these two sets.

Consider the computation of $[v]_{\overrightarrow{2e}}$. An efficient way to compute this set is based on the dominators and bridges of the flow graph $G(v)$. In particular, we compute the dominator tree $D(v)$ and identify the bridges of $G(v)$. Then, for each bridge $e = (u, w)$, we have $d(w) = u$, i.e., each bridge is also an edge in the dominator tree; we mark w in $D(v)$.

LEMMA 3.3. *$w \in [v]_{\overrightarrow{2e}}$ if and only if w is not dominated in $G(v)$ by a marked vertex.*

PROOF. We have that $w \notin [v]_{\overrightarrow{2e}}$ if and only if there is an edge (strong bridge) that separates v from w in G. Then, $e = (x, y)$ is such an edge if and only if it is a bridge in $G(v)$, so y is a marked ancestor of w in $D(v)$. □

Lemma 3.3 implies a straightforward linear-time algorithm to compute $[v]_{\overrightarrow{2e}}$, given the dominator tree $D(v)$ of $G(v)$. We use the same algorithm to compute $[v]_{\overleftarrow{2e}}$, but operate on the reverse graph $G^R(v)$ and its dominator tree $D^R(v)$. That is, we identify the bridges of the flow graph $G^R(v)$, and for each bridge $e = (u, w)$, we mark w in $D^R(v)$. Note that a vertex w that is marked in $D(v)$ may not be marked in $D^R(v)$ and vice versa.

COROLLARY 3.4. *$w \in [v]_{2e}$ if and only if w is not dominated in $G(v)$ and in $G^R(v)$ by any marked vertex. Moreover, $[v]_{2e}$ can be computed in $O(m)$ time.*

Note that all the 2-edge-connected blocks $[v]_{2e}$ can be computed in $O(mn)$ time by applying Corollary 3.4 to all vertices v. We describe next a more complicated algorithm that avoids repeated applications of Corollary 3.4. This algorithm will still require $O(mn)$ time, but it will be a useful ingredient for our linear-time algorithm.

3.2.1. Canonical Decomposition. Let s be an arbitrarily chosen start vertex. We first observe that the bridges in the dominator trees $D(s)$ and $D^R(s)$ of $G(s)$ and $G^R(s)$, respectively, partition the vertices into sets that contain the 2-edge-connected blocks. More precisely, identify the bridges of $G(s)$ (resp., $G^R(s)$), and for each bridge $e = (u, w)$, mark w in $D(s)$ (resp., $D^R(s)$), as above. Now delete all bridges from $D(s)$ and $D^R(s)$: namely, remove from $D(s)$ all edges $(d(v), v)$ such that v is marked in $D(s)$, and remove from $D^R(s)$ all edges $(d^R(v), v)$ such that v is marked in $D^R(s)$. This decomposes the dominator trees $D(s)$ and $D^R(s)$ into forests of rooted trees, where each tree is rooted either at a marked vertex or at the start vertex s. In the following, we refer to this as the *canonical decomposition* of the dominator tree $D(s)$, and use the notation $T(v)$ to denote the tree containing vertex v in this canonical decomposition. Note that $T(v)$ is a subtree of $D(s)$, and its root r_v is either a marked vertex or the start vertex s. Similarly, we denote by $T^R(v)$ the tree containing vertex v in the canonical decomposition of the dominator tree $D^R(s)$. Figure 5 shows an example of a flow graph $G(s)$, its dominator tree $D(s)$, and the canonical decomposition of $D(s)$ into the subtrees $T(v)$ induced by the removal of all bridges of the flow graph $G(s)$.

In the following lemmas, we assume that s is an arbitrarily chosen start vertex in G, $G(s)$ is the flow graph with start vertex s, $G^R(s)$ is the flow graph obtained from $G(s)$ after reversing edge directions, $D(s)$ and $D^R(s)$ are the dominator trees of $G(s)$ and $G^R(s)$, respectively, and $T(v)$ and $T^R(v)$ are the subtrees containing vertex v in the canonical decompositions of $D(s)$ and $D^R(s)$, respectively (i.e., induced by the removal of all bridges in $D(s)$ and $D^R(s)$).

LEMMA 3.5. *Let v and w be two different vertices in G. Then, $[v]_{2e} = [w]_{2e}$ only if $T(v) = T(w)$ and $T^R(v) = T^R(w)$.*

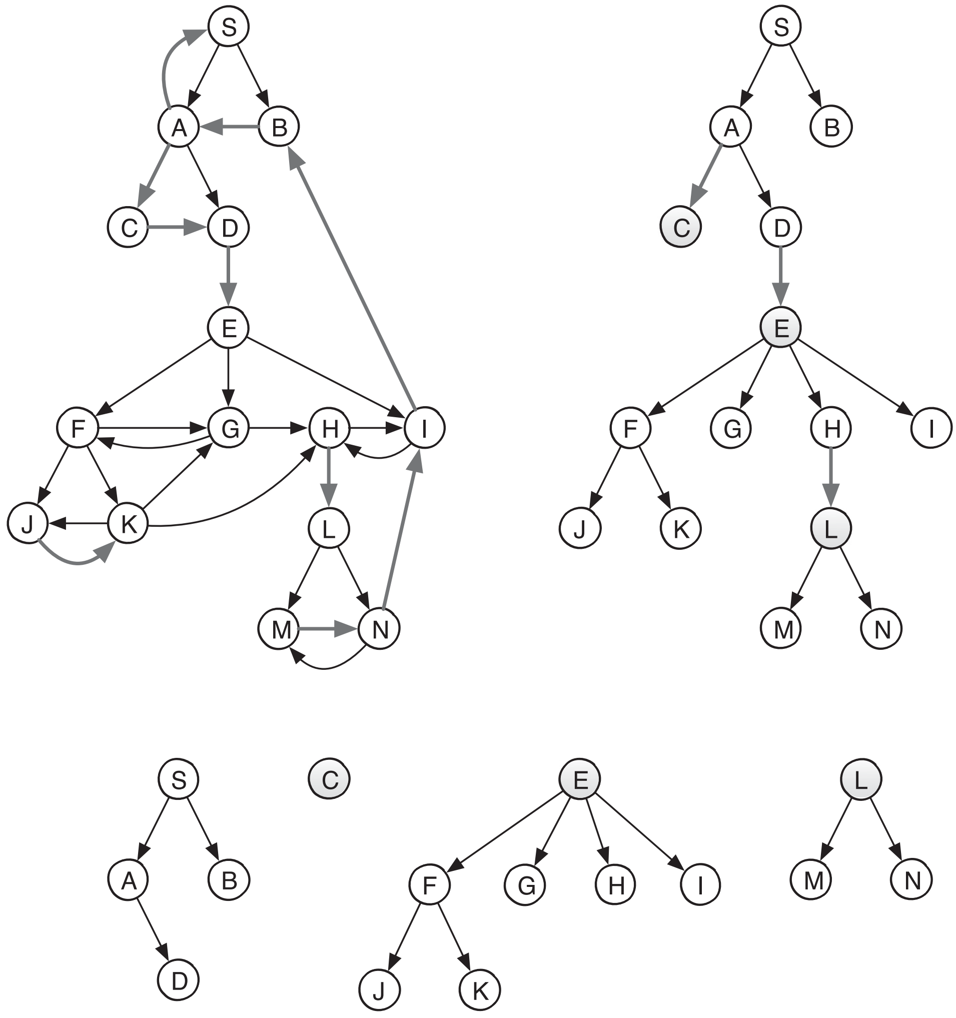

Fig. 5. A flow graph $G(s)$, its dominator tree $D(s)$, and its canonical decomposition into the subtrees $T(v)$ induced by the bridges of $G(s)$. Strong bridges of the original graph G and bridges of the flow graph $G(s)$ are shown in red; marked vertices are shown in yellow. (Better viewed in color.)

Proof. We show that $[v]_{2e} = [w]_{2e}$ implies $T(v) = T(w)$. Then, the same argument applied on $G^R(s)$ shows that $T^R(v) = T^R(w)$. Suppose, by contradiction, that $[v]_{2e} = [w]_{2e}$, but $T(v) \neq T(w)$, i.e., $w \notin T(v)$. Assume, without loss of generality, that r_v is not an ancestor of r_w in $D(s)$ (if not, swap v and w). Note that the edge $e = (d(r_v), r_v)$ must be a bridge in $G(s)$. Since $[v]_{2e} = [w]_{2e}$, then there must be a path P in G from w to v that avoids edge e. Since r_v is not an ancestor of r_w in $D(s)$, there is a path Q in G from s to w that avoids e. If $v \in Q$, then the part of Q from s to v avoids e, which contradicts the fact that e is a bridge in $G(s)$, i.e., it induces a cut that separates s from v in G. If $v \notin Q$, then Q followed by P $(Q \cdot P)$ gives a path from s to v in G that avoids the bridge e, again a contradiction. $\square$

Note that Lemma 3.5 provides a necessary condition for two vertices to be 2-edge-connected. This is not a sufficient condition, however, as two vertices may be separated

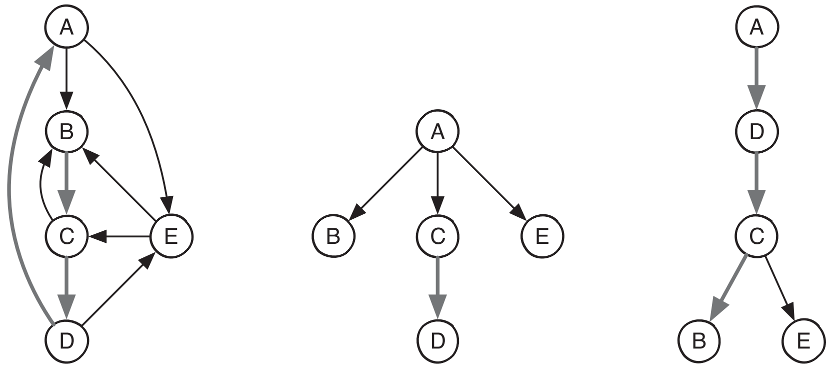

Fig. 6. A strongly connected digraph G and the dominator trees $D(A)$ and $D^R(A)$ of the corresponding flow graphs $G(A)$ and $G^R(A)$, respectively, rooted at vertex A. Strong bridges are shown in red (better viewed in color). Although vertices C and E lie in the same subtree in both the canonical decomposition of $D(A)$ and of $D^R(A)$, they are not 2-edge-connected, as they are separated by the strong bridge (C, D).

by a strong bridge and still lie in the same subtree in both the canonical decompositions of $D(s)$ and of $D^R(s)$ (see Figure 6). The main challenge in this approach is, thus, to discover which vertices in the same subtree are separated by a strong bridge. To tackle this challenge, we provide some key observations regarding edges and paths that connect different subtrees $T(r)$. We will use the *parent property* of dominator trees [Georgiadis and Tarjan 2015] that we state next.

LEMMA 3.6. *(Parent property of the dominator tree [Georgiadis and Tarjan 2015].) For all $(v, w) \in E$, $d(w)$ is an ancestor of v in $D(s)$.*

Now we prove some structural properties for paths that connect vertices in different subtrees.

LEMMA 3.7. *Let $e = (u, v)$ be an edge of G such that $T(u) \neq T(v)$, and let r_v be the root of $T(v)$. Then, either $u = d(v)$ and e is a bridge in the flow graph $G(s)$, or u is a proper descendant of r_v in $D(s)$.*

PROOF. If e is a bridge in $G(s)$, then $u = d(v)$ and the lemma holds. Suppose that e is not a bridge, so $u \neq d(v)$. If v is an ancestor of u in $D(s)$, then the lemma holds. If not, then by Lemma 3.6, $d(v)$ is a proper ancestor of u in $D(s)$. We show that $d(v) \in T(v)$, which implies the lemma. Assume, by contradiction, that $d(v) \notin T(v)$. Then, $(d(v), v)$ is a bridge and $v = r_v$. Since v is not an ancestor of u in $D(s)$, there is a path P from s to u that does not contain v. Then, $P \cdot e$ is a path from s to v that avoids the bridge $(d(v), v)$, a contradiction. □

LEMMA 3.8. *Let r be a marked vertex in $D(s)$. Let v be any vertex that is not a descendant of r in $D(s)$. Then, there is a path from v to r that does not contain any vertex in $T(r) \setminus r$. Moreover, all simple paths from v to any vertex in $T(r)$ contain the edge $(d(r), r)$.*

PROOF. Since v is not a descendant of r in $D(s)$, $v \notin T(r)$. Graph G is strongly connected, so it contains a path from v to r. Let P be any such path. Let $e = (u, w)$ be the first edge on P such that $w \in T(r)$. Then, by Lemma 3.7, either $e = (d(r), r)$ or u is a proper descendant of r. In the first case, the lemma holds. Suppose u is a proper descendant of r. Since v is not a descendant of r in $D(s)$, there is a path Q from s to v in G that does not contain r. Then Q, followed by the part of P from v to w, gives a path from s to $w \in T(r)$ that avoids $d(r)$, a contradiction. □

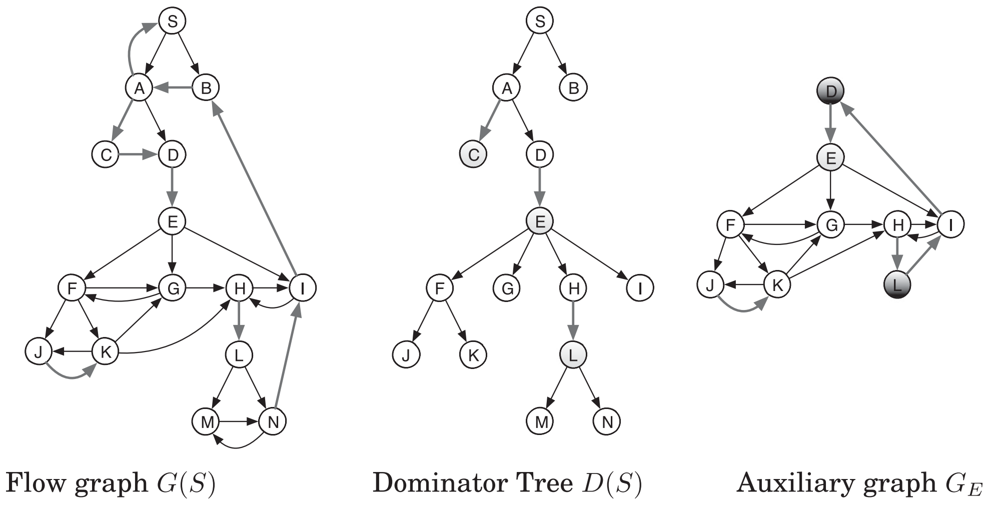

Fig. 7. The flow graph $G(S)$ and its dominator tree $D(S)$ from Figure 5, together with the auxiliary graph of vertex E. Strong bridges are red, marked vertices are yellow, and auxiliary vertices are gray. (Better viewed in color.)

3.2.2. Auxiliary Graphs. We now introduce the notion of *auxiliary graphs* that plays a crucial role in our algorithm. Auxiliary graphs represent a decomposition of the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original 2-edge-connected blocks of G. Let r be either a non-leaf marked vertex or the start vertex s in the dominator tree $D(s)$, and let $T(r)$ be the subtree with root r in the canonical decomposition of the dominator tree $D(s)$. For each such subtree $T(r)$, we define the *auxiliary graph* $G_r = (V_r, E_r)$ *of* r as follows. The vertex set V_r of G_r consists of all the vertices in $T(r)$, referred to as *ordinary* vertices, and a set of *auxiliary* vertices, which are obtained by contracting vertices in $V \setminus T(r)$, as follows. Let v be a vertex in $T(r)$. We say that v is a *boundary vertex in* $T(r)$ if v has a marked child in $D(s)$. Let w be a marked child of a boundary vertex v: All the vertices that are descendants of w in $D(s)$ are contracted into w. All vertices in $V \setminus T(r)$ that are not descendants of r are contracted into $d(r)$ ($r \neq s$ if any such vertex exists). During those contractions, parallel edges are eliminated. This is necessary in order to obtain the size bound given in Lemma 3.9. Figure 7 shows a flow graph, its dominator tree, and an auxiliary graph.

LEMMA 3.9. *If $G(s)$ has b bridges, then all the auxiliary graphs G_r have at most $n+2b$ vertices and $m+2b$ edges in total.*

PROOF. Every vertex appears as an ordinary vertex only in one auxiliary graph. A marked vertex in $D(s)$ corresponds to a bridge in $G(s)$, so there are $b \leq n-1$ marked vertices. Since we have one auxiliary graph for each marked vertex, the total number of the auxiliary vertices $d(r)$ is b. Each marked vertex v can also appear in at most one other auxiliary graph as a child of a boundary vertex. So, the total number of vertices is at most $n+2b$. Next, we bound the total number of edges. Excluding bridges, the total number of edges between two ordinary vertices or between an ordinary vertex and an auxiliary vertex in all auxiliary graphs is at most $m-b$. Each bridge can appear in at most two auxiliary graphs. It remains to count the number of edges between two auxiliary vertices. Each such edge is of the form $(w, d(r))$, where w and r are roots in the canonical decomposition, and moreover, w is a marked child of a boundary vertex in the auxiliary graph G_r. Hence, there is at most one such edge leaving each marked vertex, so at most b overall. The total number of edges in all auxiliary graphs is then bounded by $m - b + 2b + b = m + 2b$. □

LEMMA 3.10. *Let v and w be two vertices in a subtree $T(r)$. Any path P from v to w in G has a corresponding path P_r from v to w in the auxiliary graph G_r, and vice versa. Moreover, all paths from v to w in G contain a common strong bridge of G if and only if all paths from v to w in G_r contain a common strong bridge of G_r.*

PROOF. The first part of the lemma follows immediately from the definition of the auxiliary graph G_r.

For the second part, first note that all bridges of $G(s)$ that are edges in an auxiliary graph G_r are strong bridges of G_r. Specifically, if $r \neq s$, then $(d(r), r)$ is the only edge leaving the auxiliary vertex $d(r)$ in G_r, so it is a strong bridge in G_r. Similarly, a bridge $(d(z), z)$ of $G(s)$ with $d(z) \in T(r)$ is a strong bridge in G_r, since it is the only edge entering the auxiliary vertex z in G_r. Now suppose that all paths from v to w in G contain a strong bridge $e = (x, y)$ of G. We consider the following cases:

(i) $x \in T(r)$ and $y \in T(r)$. Then, by the construction of G_r, (x, y) is also an edge on every path from v to w in G_r. Hence, (x, y) is a strong bridge also in G_r.

(ii) $x \in T(r)$ and $y \notin T(r)$. By Lemma 3.7, either $x = d(y)$ or x is a proper descendant of r_y in the dominator tree $D(s)$. In the former case, $e = (d(y), y)$ and the definition of G_r implies that e is contained in all paths from v to w in G_r. In the latter case, r_y is not a descendant of r in $D(s)$. Hence, all paths from v to w in G contain a vertex that is not a descendant of r in $D(s)$, namely y. Then, Lemma 3.8 implies that all paths from v to w in G contain the strong bridge $(d(r), r)$. By the construction of G_r, $(d(r), r)$ is also an edge on every path from v to w in G_r.

(iii) $x \notin T(r)$ and x is a descendant of r in $D(s)$. We claim that v is not an ancestor of w in $D(s)$. We prove the claim by contradiction. So, assume that v is an ancestor of w in $D(s)$. Since v and w are both ordinary vertices of G_r, e is not on the path from v to w in $D(s)$. But then, there is a path from v to w in G that avoids e, a contradiction. So, v is not an ancestor of w in $D(s)$. Let $t \in T(r)$ be the boundary vertex that is an ancestor of x, and let z be the child of t that is an ancestor of x in $D(s)$. Since v is not a descendant of x in $D(s)$, all paths from v to x in G contain z. Otherwise, there would be a path in G from s to x through v that avoids z, contradicting the fact that z is an ancestor of x in $D(s)$. By Lemma 3.8, all paths from v to z in G, and thus, all paths from v to w, contain the strong bridge $(t, z) = (d(z), z)$. By the construction of G_r, (t, z) is also an edge on every path from v to w in G_r.

(iv) $x \notin T(r)$ and x is not a descendant of r in $D(s)$. In this case, Lemma 3.8 implies that all paths from x to w in G contain the strong bridge $(d(r), r)$. Hence, all paths from v to w in G contain the strong bridge $(d(r), r)$, and so do all paths from v to w in G_r by the construction of the auxiliary graphs.

Suppose now that all paths from v to w in G_r contain a strong bridge $e = (x, y)$ of G_r. We consider the following cases:

(i) x and y are both ordinary vertices in G_r. By the definition of the auxiliary graphs, we have that all paths from v to w in G contain e.

(ii) x is an ordinary vertex and y is an auxiliary vertex in G_r. Then, $y \neq w$, and either $y = d(r)$ or $x = d(y)$. In the former case, all paths from v to w in G_r must contain $(d(r), r)$, since $d(r) \neq w$ and $(d(r), r)$ is the unique edge leaving $d(r)$ in G_r. Similarly, if $x = d(y)$, then e is a bridge in $G(s)$ and, by the definition of the auxiliary graphs, all paths from v to w in G contain e.

(iii) x is auxiliary vertex and y is ordinary vertex in G_r. Then, either $x = d(r)$ or $(d(x), x)$ is a bridge in $G(s)$. By the same arguments as in case (ii), the former implies that all paths from v to w in G contain $(d(r), r)$, and the latter implies that all paths from v to w in G contain $(d(x), x)$.

(iv) x and y are both auxiliary vertices in G_r. Then, $y = d(r)$, and the same argument as in case (ii) implies all paths from v to w in G contain $(d(r), r)$.

Therefore, we conclude that all paths from v to w in G contain a common strong bridge of G if and only if all paths from v to w in G_r contain a common strong bridge of G_r. □

COROLLARY 3.11. *Each auxiliary graph G_r is strongly connected.*

PROOF. It follows immediately from Lemma 3.10 and the fact that G is strongly connected. □

Now we are ready to show that we can compute the 2-edge-connected blocks in each auxiliary graph independently of each other.

LEMMA 3.12. *Let v and w be any two distinct vertices of G. Then, v and w are 2-edge-connected in G if and only if they are both ordinary vertices in an auxiliary graph G_r, where r is a root in the canonical decomposition of $D(s)$, and they are 2-edge-connected in G_r.*

PROOF. By Lemma 3.5, v and w are 2-edge-connected in G only if they belong to the same subtree $T(r)$, where r is a root in the canonical decomposition of $D(s)$. In this case, both v and w are ordinary vertices of G_r. If v and w are 2-edge-connected in G, then Lemma 3.10 implies that they are also 2-edge-connected in G_r. Now suppose that there is a strong bridge that separates v from w in G. By Lemma 3.10 we have that all paths from v to w in G_r also have a strong bridge in common. The same argument applies if there is a strong bridge that separates w from v in G, and the lemma follows. □

Fast Construction of Auxiliary Graphs. We next show how to construct the auxiliary graphs $G_r = (V_r, E_r)$ efficiently. The vertex set V_r contains the set V_r^o of *ordinary* vertices (i.e., the vertices of $T(r)$) and the set V_r^a of *auxiliary* vertices. The edge set E_r contains all edges in $G = (V, E)$ induced by the ordinary vertices (i.e., edges $(u, v) \in E$ such that $u \in T(r)$ and $v \in T(r)$), together with some edges that have at most one endpoint in $T(r)$ and are either bridges of $G(s)$ or *shortcut* edges that correspond to paths in G. We define shortcut edges as follows. Let v be a *boundary vertex in $T(r)$* (i.e., v has a marked child in $D(s)$). For each marked child w of v in $D(s)$, we add a copy of w in V_r^a and add the edge (v, w) in E_r. Also, if r is marked ($r \neq s$), then we add a copy of $d(r)$ in V_r^a, and add the edge $(d(r), r)$ in E_r. We also add in E_r the following shortcut edges for edges (u, v) of the following type:

(a) If u is ordinary and v is not a descendant of r, then we add the shortcut edge $(u, d(r))$.
(b) If v is ordinary and u is a proper descendant in $D(s)$ of a boundary vertex w, then we add the shortcut edge (z, v), where z is the child of w that is an ancestor of u in $D(s)$.
(c) Finally, if u is a proper descendant in $D(s)$ of a boundary vertex w and v is not a descendant of r, then we add the shortcut edge $(z, d(r))$, where z is the child of w that is an ancestor of u in $D(s)$.

We note that, according to the definition, in this construction, we do not keep multiple (parallel) shortcut edges (see Figure 7).

To complete our construction of the auxiliary graphs, we need to specify how to compute the shortcut edges of each type (a), (b), and (c). Suppose (u, v) is an edge of type (a). Then, v is not a descendant of r in $D(s)$, which can be tested using an $O(1)$-time test of the ancestor-descendant relation. There are several simple $O(1)$-time tests of this relation [Tarjan 1974b]. The most convenient one for us is to number the vertices of $D(s)$ from 1 to n in pre-order and compute the number of descendants of each vertex v; we denote these numbers by $pre(v)$ and $size(v)$, respectively. Then, v is a descendant of r if and only if $pre(r) \leq pre(v) < pre(r) + size(r)$.

Next, suppose that (u, v) is of type (b). Then, u is a proper descendant of a boundary vertex w in $D(s)$. To compute the shortcut edge of (u, v), we need to find the child z of w that is an ancestor of u in $D(s)$. To that end, we create a list B_r that contains the edges (u, v) of type (b) such that $v \in T(r)$, and sort B_r in increasing pre-order of u. We create a second list B'_r that contains the children in $D(s)$ of the boundary vertices in $T(r)$, and sort B'_r in increasing pre-order. Then, the shortcut edge of (u, v) is (z, v), where z is the last vertex in the sorted list B'_r such that $pre(z) \leq pre(u)$. Thus, the shortcut edges of type (b) can be computed in linear time by bucket sorting and merging. In order to compute all type (b) edges in linear time, we sort all the lists at the same time as follows. First, we create a unified list B containing the triples $(r, pre(u), v)$, for each type (b) edge (u, v) in the auxiliary graph G_r. Next, we sort B by the first two elements in each triple, so that for any two consecutive triples $(r_1, pre(u_1), v_1)$ and $(r_2, pre(u_2), v_2)$, either $r_1 < r_2$, or $r_1 = r_2$ and $pre(u_1) \leq pre(u_2)$. We also create a second list B' with pairs $(r, pre(z))$, where z is a child of a boundary vertex $w \in T(r)$, and sort the pairs as in B. Finally, we compute the shortcut edges of each auxiliary graph G_r by merging the sorted sublists of B and B' that correspond to the same root r. Then, the shortcut edge for the triple $(r, pre(u), v)$ is (z, v), where $(r, pre(z))$ is the last pair in the sorted sublist of B' with root r such that $pre(z) \leq pre(u)$.

Finally, consider the edges of type (c). For each such edge (u, v), we need to add the edge $(z, d(r))$ in each G_r, where u is a proper descendant of a boundary vertex $w \in T(r)$, v is not a descendant of r in $D(s)$, and z is the child of w that is an ancestor of u in $D(s)$. We compute these edges for all auxiliary graphs G_r as follows. First, we create a compressed tree $\widehat{D}(s)$ that contains only s and the marked vertices. A marked vertex v becomes child of its nearest marked ancestor u, or of s if u does not exist. This can be easily done in $O(n)$ time during the pre-order traversal of $D(s)$. Next, we process all edges (u, v) such that v is not a descendant of r_u in $D(s)$. At each node $w \neq s$ in $\widehat{D}(s)$, we store a label $\ell(w)$ which is the minimum $pre(r_v)$ of an edge (u, v) of type (c) such that $u \in T(w)$; we let $\ell(w) = pre(w)$ if no such edge exists. Using these labels, we compute for each $w \neq s$ in $\widehat{D}(s)$ the values $low(w) = \min\{\ell(v) \mid v \text{ is a descendant of } w \text{ in } \widehat{D}(s)\}$. These computations can be done in $O(m)$ time by processing the tree $\widehat{D}(s)$ in a bottom-up order. Now consider the auxiliary graph G_r. We process the children in $D(s)$ of the boundary vertices in $T(r)$. Note that these children are marked, so they have a *low* value. For each such child z we test if G_r has a shortcut edge $(z, d(r))$: If $low(z) < pre(r)$, then we add the edge $(z, d(r))$. This leads to the following lemma.

LEMMA 3.13. *We can compute all auxiliary graphs G_r in $O(m)$ time.*

We note that, alternatively, we can compute the type (b) shortcut edges of the auxiliary graphs, by replacing sorting with vertex contractions. To that end, we use a *disjoint set union data structure* [Tarjan 1975] which maintains a collection of disjoint sets, each with a representative element, under three operations:

make-set(x):	Create a new set $\{x\}$ with representative x. Element x must be in no existing set.
find(x):	Return the representative element of the set containing element x.
unite(x, y):	Unite the sets containing elements x and y and give the new set the representative of the old set containing x.

Algorithm Rec2ECB: Recursive computation of the 2-edge-connected blocks for the ordinary vertices of a strongly connected digraph $H = (V, E)$

Step 1: Choose an arbitrary ordinary vertex $s \in V^o$ as a start vertex. Compute the dominator trees $D(s)$ and $D^R(s)$ and the bridges of the flow graphs $H(s)$ and $H^R(s)$.

Step 2: Compute the number b of bridges (x, y) in $H(s)$ such that y is an ancestor of an ordinary vertex in $D(s)$. Compute the number b^R of bridges (x, y) in $H^R(s)$ such that y is an ancestor of an ordinary vertex in $D^R(s)$.

Step 3: If $b = b^R = 0$ then return $[s]_{2e} = V^o$.

Step 4: If $b^R > b$ then swap H and H^R. Find the canonical decomposition of $D(s)$ into the subtrees $T(r)$ and compute the corresponding auxiliary graphs H_r. Compute recursively the 2-edge-connected blocks for each auxiliary graph H_r with at least two ordinary vertices.

Fig. 8. Algorithm Rec2ECB.

Using this data structure, we can compute the type (b) shortcut edges of all auxiliary graphs in a single bottom-up pass of $D(s)$. To initialize the sets, we perform *make-set*(v) for every vertex $v \in V$. The unite operations are also executed in a bottom-up order so that the following invariant is maintained. Suppose that we process a root vertex r of the canonical decomposition, i.e., r is marked or $r = s$. For each strong bridge (w, z) such that w is a boundary vertex of $T(r)$, all descendants x of z in $D(s)$ are contracted into z, i.e., *find*$(x) = z$. The details of these computations are as follows. We process all roots r in a bottom-up order of $D(s)$ and compute the type (b) shortcut edges of the corresponding auxiliary graph $G_r = (V_r, E_r)$. To do this, we process the edges entering each vertex $v \in T(r)$. For each such edge (u, v), we compute $z =$ *find*(u). If $z \notin T(r)$ and $d(z) \in T(r)$, then (z, v) is a type (b) shortcut edge of G_r. After we have processed all edges entering $T(r)$, we execute *unite*(r, v) for each vertex $v \in T(r)$ and each vertex $v \in V_r^a$ that is a child in $D(s)$ of a boundary vertex in $T(r)$. This construction runs in $O(m)$ time plus the time for at most $n-1$ *unite* operations and, by Lemma 3.9, less than $m + 2n$ *find* operations. If *unite* and *find* are implemented using compressed trees with balanced linking [Tarjan 1975], the total time for these disjoint set union operations is $O(m\alpha(m, n))$. Alternatively, since the set of *unite* operations is known in advance, we can substitute the operations *unite*(r, v) with *unite*$(d(v), v)$ so that we have an instance of the static tree disjoint set union problem, which is solvable in $O(m)$ time on a RAM [Gabow and Tarjan 1985].

3.3. A Recursive Algorithm

Lemma 3.12 allows us to compute the 2-edge-connected blocks of each auxiliary graph separately. Algorithm Rec2ECB, described in Figure 8, applies this idea recursively until all ordinary vertices in each graph H are 2-edge-connected. Since, by Lemma 3.5, an auxiliary vertex and an ordinary vertex of H are not 2-edge-connected, we only need to consider the strong bridges that separate ordinary vertices of H. In order to find if H contains such strong bridges, we choose an arbitrary ordinary vertex s of H as a start vertex and compute the bridges and the dominator trees $D(s)$ and $D^R(s)$ of the flow graphs $H(s)$ and $H^R(s)$, respectively. Then, Corollary 3.4 implies that H contains a strong bridge that separates two ordinary vertices if and only if $H(s)$ or $H^R(s)$ contains a bridge (x, y) such that y is an ancestor of an ordinary vertex. Otherwise, all ordinary vertices are 2-edge-connected to and from s, so $[s]_{2e} = V^o$. We remark that in the first call of Algorithm Rec2ECB, the input graph is G and all the vertices are considered ordinary, i.e., $H = G$ and $V^o = V$. We also note that in Step 4, we actually need to swap

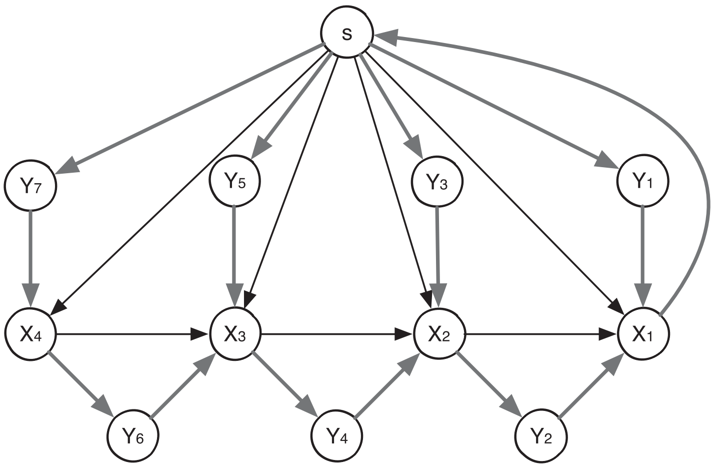

Fig. 9. An input digraph with $n = \Theta(k)$ vertices that causes k recursive calls of Algorithm Rec2ECB (See Figures 10 and 11). Vertices $X_1, X_2, \ldots, X_k$ are not 2-edge-connected, but Algorithm Rec2ECB requires k recursive calls to separate them into different blocks. (In this figure, $k = 4$.) Each new partition is induced by the strong bridge (X_1, S).

the role of H and H^R only when $b = 0$ and $b^R \neq 0$. This is required so that the algorithm makes progress in separating ordinary vertices that belong to different blocks. By doing this swap when $b^R > b$, one may hope to reduce the number of recursive calls made by the algorithm. But, as the example of Figure 9 shows, the number of recursive calls can still be $O(n)$ in the worst case.

LEMMA 3.14. *Algorithm Rec2ECB runs in $O(mn)$ time.*

PROOF. Each recursive call refines the current partition of V; thus, we have at most $n-1$ recursive calls. By Buchsbaum et al. [2008], Tarjan [1974a], and Lemma 3.13, the total work per recursive call is $O(m)$. □

We note that the bound stated in Lemma 3.14 is tight. The same strong bridge can be used repeatedly to separate different pairs of vertices in successive recursive calls (see Figures 9, 10, and 11). Despite the fact that Algorithm Rec2ECB only achieves an $O(mn)$ time bound, it will be the basis of our linear-time algorithm that we develop in the next section.

3.4. A Linear-Time Algorithm

Although Algorithms Simple2ECB and Rec2ECB run in $O(mn)$ time, we show that a careful combination of them gives a linear-time algorithm. The critical observation, proved in Lemma 3.15 below, is that if a strong bridge separates different pairs of vertices in successive recursive calls (which causes the worst-case behavior of Algorithm Rec2ECB, as shown in Figures 9, 10, and 11), then it will appear as the strong bridge entering the root of a subtree in the canonical decomposition of a dominator tree.

Algorithm Fast2ECB, described in Figure 12, applies the observation above together with all the tools we developed in the previous sections and achieves the computation of the 2-edge-connected blocks in linear time. In essence, it runs Algorithm Rec2ECB but stops the recursion at depth 2. Thus, it uses only two levels of auxiliary graphs, as shown in Figure 13. Two vertices that are not 2-edge-connected but have not been separated yet, i.e., they are ordinary vertices of an auxiliary graph computed at recursion depth 2, can be separated by running Algorithm Simple2ECB for the specific

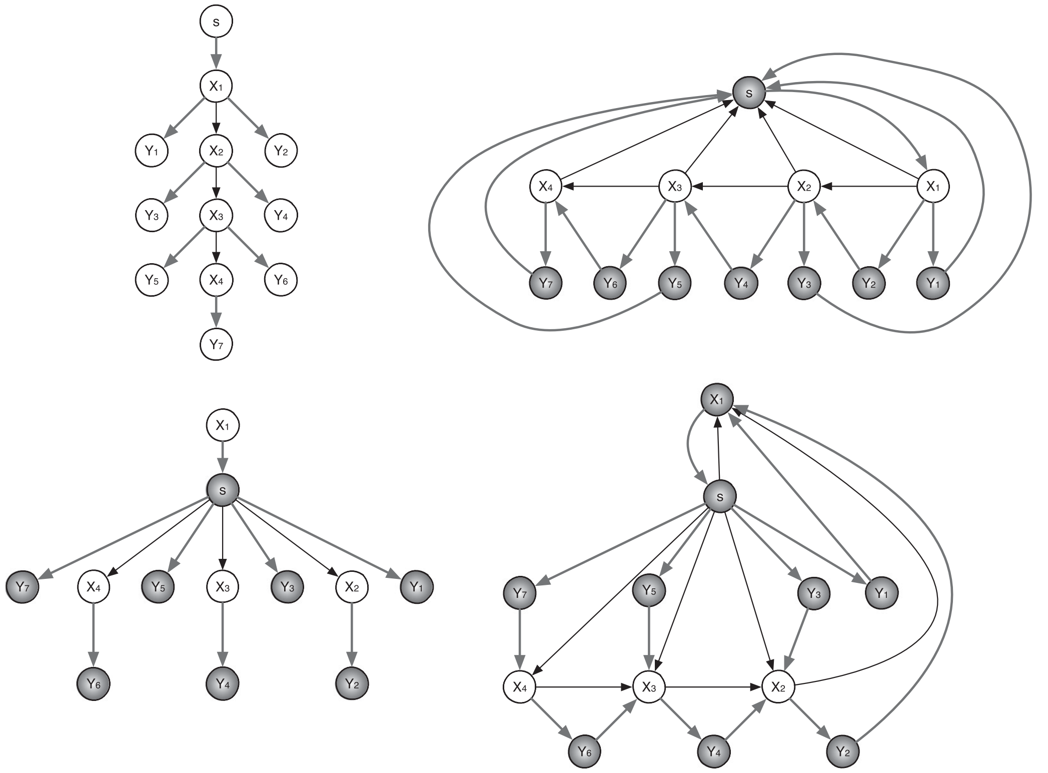

Fig. 10. The first two recursive calls of Algorithm Rec2ECB, with input from the digraph of Figure 9. In the left column, we see the dominator tree used to compute the next partition, while in the right column, there is the auxiliary graph containing the majority of ordinary vertices which will be the input digraph in the next recursive call.

auxiliary graph. However, as we show in the proof of Lemma 3.15, it suffices to remove only one strong bridge of that auxiliary graph, so we only need to execute Step 3 of Algorithm Simple2ECB once. Figure 13 illustrates the main operations of the algorithm, and Figure 14 gives an example of the execution of Step 3.

LEMMA 3.15. *Algorithm Fast2ECB is correct.*

PROOF. Let u and v be any vertices. If u and v are 2-edge-connected in G, then by Lemma 3.12, u and v are located as ordinary vertices in the same auxiliary graph $H = G_r$ of G, and they are 2-edge-connected in H. By applying Lemma 3.12 on H^R, we also have that u and v are located as ordinary vertices in the same auxiliary graph H_q^R of H^R, and they are 2-edge-connected in H_q^R. This implies that the algorithm will correctly include them in the same block. So, suppose that u and v are not 2-edge-connected. Then, without loss of generality, we can assume that all paths from u to v contain a common strong bridge. We argue that the blocks of u and v will be separated in some step of the algorithm.

If u and v are located in different subtrees of $D(s)$, then the claim is true. If they are in the same subtree, then they appear in an auxiliary graph $H = G_r$ as ordinary vertices. By Lemma 3.12, H contains a strong bridge that is contained in all paths from u to v. Let H^R be the reverse graph of H. Let $D_H^R(r)$ be the dominator tree of $H^R(r)$. If u and v are located in different subtrees of D_H^R, then the claim is true. Suppose, then,

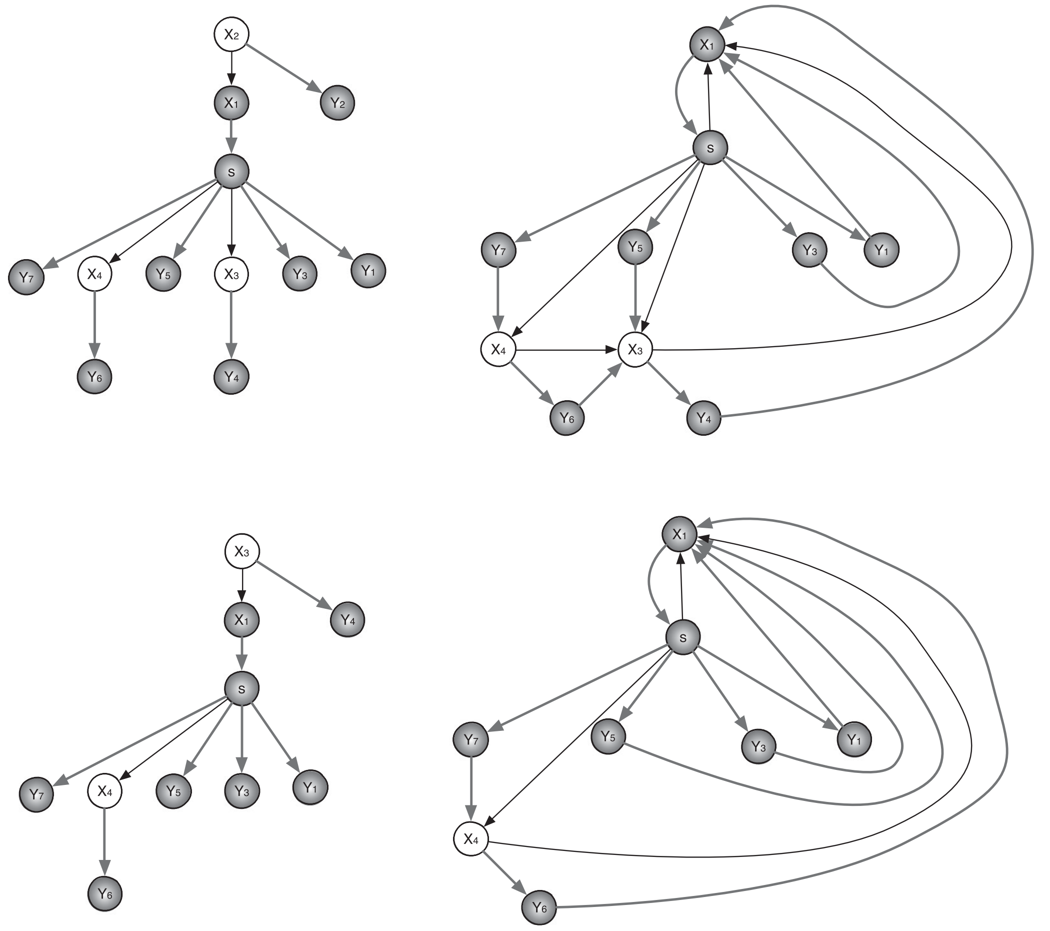

Fig. 11. The other two recursive calls of Algorithm Rec2ECB, with input from the digraph of Figure 9 (the first two calls are shown in Figure 10). In the left column, we see the dominator tree used to compute the next partition, while in the right column, there is the auxiliary graph containing the majority of ordinary vertices which will be the input digraph in the next recursive call.

that they are located in a subtree with root q. By Corollary 3.4, $q \neq r$. Let $p = d_H^R(q)$ be the parent of q in $D_H^R(r)$. Then, (q, p) is a strong bridge of H. (See Figure 13). We claim that $H \setminus (q, p)$ does not contain any path from u to v.

To prove the claim, we consider two cases:

(1) All paths from v to u in H^R contain a bridge $(d_H^R(x), x)$ of $D_H^R(r)$ such that x is ancestor of u in $D_H^R(r)$. We argue that (p, q) appears in all paths from v to u in H^R. Suppose, for contradiction, that this is not true, i.e., there is a path π in H^R from v to u that avoids (p, q). Then, π contains $(d_H^R(x), x)$, by the assumption that $(d_H^R(x), x)$ appears in all paths from v to u in H^R. Since x is an ancestor of p in $D_H^R(r)$, there is a path π' in H^R from r to x that avoids q. So, π' does not contain (p, q). But then, $\pi' \cdot \pi$ is a path from r to u in H^R that avoids (p, q), a contradiction.

(2) There is no bridge $(d_H^R(x), x)$ of $D_H^R(r)$ with x as an ancestor of u in $D_H^R(r)$ that is contained in all paths from v to u in H^R. Let e be a strong bridge that separates

Algorithm Fast2ECB: Linear-time computation of the 2-edge-connected blocks of a strongly connected digraph $G = (V, E)$

```
Step 1: Choose an arbitrary vertex $s \in V$ as a start vertex. Compute the dominator
    tree $D(s)$ and the bridges of the flow graph $G(s)$.
Step 2: Partition $D(s)$ into subtrees $T(r)$ and compute the corresponding auxiliary
    graphs $G_r$.
Step 3: For each auxiliary graph $H = G_r$ do:
  Step 3.1: Compute the dominator tree $D_H^R(r)$ and the bridges of $H^R(r)$. Let $d_H^R(q)$ be
      the parent of $q \neq r$ in $D_H^R(r)$.
  Step 3.2: Partition $D_H^R(r)$ into the subtrees $T_H^R(q)$. Compute the corresponding aux-
      iliary graphs $H_q^R$ with $q \neq r$.
  Step 3.3: Set $[r]_{2e}$ to consist of the ordinary vertices in $T_H^R(r)$.
  Step 3.4: For each auxiliary graph $H_q^R$ with $q \neq r$ do:
    Step 3.4.1: Compute the strongly connected components $S_1, S_2, \ldots, S_k$ of $H_q^R \setminus$
        $(d_H^R(q), q)$.
    Step 3.4.2: Partition the ordinary vertices of $H_q$ into blocks according to each $S_j$,
        $j = 1, \ldots, k$; For each ordinary vertex $v$, $[v]_{2e}$ contains the ordinary vertices
        in the strongly connected component of $v$.
```

Fig. 12. Algorithm Fast2ECB.

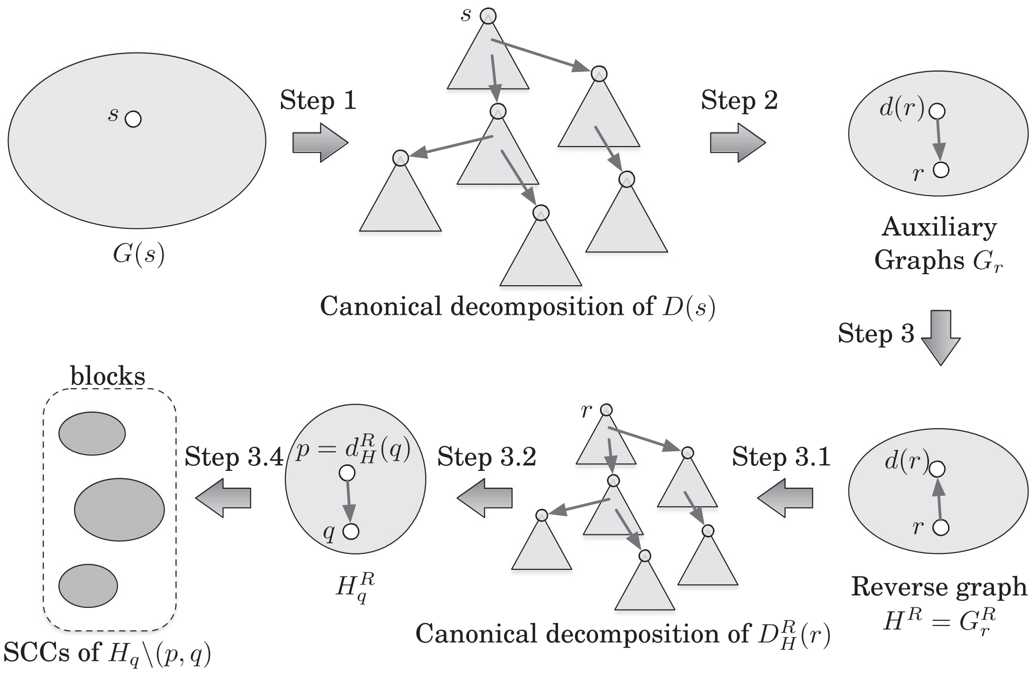
s
$G(s)$
Step 1
s
Canonical decomposition of $D(s)$
Step 2
$d(r)$
r
Auxiliary Graphs G_r
Step 3
$d(r)$
r
Reverse graph $H^R = G_r^R$
Step 3.1
r
Canonical decomposition of $D_H^R(r)$
Step 3.2
$p = d_H^R(q)$
q
H_q^R
Step 3.4
blocks
SCCs of $H_q \setminus (p, q)$

Fig. 13. A high level view of Algorithm Fast2ECB (better viewed in color).

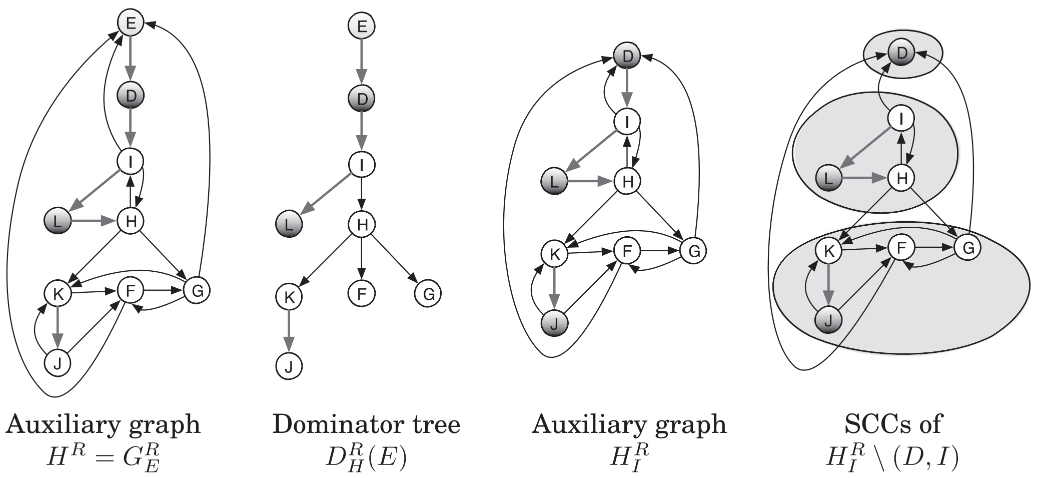

Fig. 14. The reverse graph H^R of the auxiliary graph $H = G_E$ of Figure 7 and its dominator tree $D^R_H(E)$; also, the second level auxiliary graph H^R_I and its strongly connected components after removing the strong bridge (D, I).

u from v in H. Then, $e \neq (q, p)$. The assumption that x is not an ancestor of u in $D^R_H(r)$ implies that there is a path π in H from u to r that avoids x. Hence, π does not contain e. Also, since v is an ordinary vertex of $H = G_r$, no single edge of G is contained in all paths from r to v in G. By the construction of the auxiliary graphs, there is also no edge in H that is contained in all paths from r to v in H. So, there is a path π' in H from r to v that avoids e. Then, $\pi \cdot \pi'$ is a path from u to v in H that does not contain e, a contradiction.

This shows that there is not a path from u to v in $H \setminus (q, p)$, and the lemma follows. □

Finally, we show that the algorithm, indeed, runs in linear time.

LEMMA 3.16. *Algorithm Fast2ECB runs in $O(m)$ time.*

PROOF. We analyze the total time spent on each step that Algorithm Fast2ECB executes. Step 1 takes $O(m)$ time by Buchsbaum et al. [2008], and Step 2 takes $O(m)$ time by Lemma 3.13. From Lemma 3.9, we have that the total number of vertices and the total number of edges in all auxiliary graphs H of G are $O(n)$ and $O(m)$, respectively. Therefore, the total number of strong bridges in these auxiliary graphs is $O(n)$ by Lemma 2.1. Then, by Lemma 3.9, the total size (number of vertices and edges) of all auxiliary graphs H^R_q for all H, computed in Step 3.2, is still $O(m)$, and they are also computed in $O(m)$ total time by Lemma 3.13. So, Steps 3.1 and 3.4 take $O(m)$ time in total, as well. □

4. 2-EDGE-CONNECTIVITY QUERIES

Since the 2-edge-connected blocks form a partition of the vertices of the digraph, it is straightforward to check, in constant time, if $v \leftrightarrow_{2e} w$ for any two query vertices v and w. Here, we show that if v and w are not 2-edge-connected, then we can report a witness of this fact, that is, a strong bridge e such that v and w are not in the same strongly connected component of $G \setminus e$. Using this witness, it is straightforward to verify in $O(m)$

time that v and w are not 2-edge-connected; it suffices to check that v is not reachable from w in $G \setminus e$ or vice versa.[2]

We can find the witness fast by applying Lemmas 3.5 and 3.8, which use information computed during the execution of FastECB. We do that as follows. First, consider the simpler case where $v = s$. If Lemma 3.5 does not hold for s and w in $D(s)$, then $e = (d(r_w), r_w)$ (where r_w is the root of the tree $T(w)$ in the canonical decomposition of $D(s)$) is a strong bridge that separates s from w. Otherwise, $s = r_w$, and s and w are both ordinary vertices in the auxiliary graph $H = G_s$. Then, s and w cannot satisfy Lemma 3.5 in $D^R_H(s)$, so $e = (r^R_w, d^R_H(r^R_w))$ is a strong bridge that separates w from s, where r^R_w is the root of the tree $T^R(w)$ in the canonical decomposition of $D^R_H(s)$. Now consider the case where $v, w \in V \setminus s$. Suppose first that v and w do not satisfy Lemma 3.5 in $D(s)$. Then, r_w is not an ancestor of v or r_v is not an ancestor of w (or both). Assume, without loss of generality, that r_w is not an ancestor of v. By Lemma 3.8, all paths from v to w pass through $(d(r_w), r_w)$, so $(d(r_w), r_w)$ is a strong bridge that separates v from w. On the other hand, if Lemma 3.5 holds for v and w in $D(s)$, then v and w are both ordinary vertices in an auxiliary graph $H = G_r$, where $r = r_v = r_w$. Since v and w are not 2-edge-connected in G, Lemma 3.12 implies that v and w are not 2-edge-connected in H. If they violate Lemma 3.5 for $D^R_H(r)$, then we can find a strong bridge that separates them, as above. Finally, assume that Lemma 3.5 holds for v and w in $D^R_H(r)$. Now v and w are both ordinary vertices in an auxiliary graph H^R_q. From the proof of Lemma 3.15, we have that $(q, d^R_H(q))$ is a strong bridge that separates v and w.

Observe that a separating strong bridge e found by this method may not be a real edge of the input graph; so in this case, we need to return an appropriate original strong bridge. Note that if e is a separating auxiliary edge, then it is a shortcut edge of a first-level auxiliary graph. So, we can associate each such edge e with an equivalent original strong bridge during the construction of the first-level auxiliary graphs. We do that as follows. Suppose that $e = (u, z)$ is a first-level auxiliary edge that separates v from w, where both v and w are in the same canonical subtree $T(r)$. If e is of type (a) or (c), then $z = d(r)$, and by Lemma 3.8, $(d(r), r)$ is an original strong bridge that separates v from w. Similarly, if e is of type (b), then $d(u)$ is a boundary vertex and $(d(u), u)$ is an original strong bridge that separates v from w.

All the above tests can be performed in constant time. It suffices to store the dominator tree $D(s)$ of $G(s)$ and the dominator trees $D^R_H(r)$ of all auxiliary graphs $H^R = G^R_r$. The space required for these data structures is $O(n)$ by Lemma 3.9.

5. SPARSE CERTIFICATE FOR THE 2-EDGE-CONNECTED BLOCKS

We now show how to compute in linear time a sparse certificate for the 2-edge-connected blocks, i.e., a subgraph $C(G)$ of the input graph G that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph. Such a sparse certificate allows us to speed up computations, such as finding the actual edge-disjoint paths that connect a pair of vertices (see, e.g., Nagamochi and Ibaraki [1992]). As in Section 3 we can assume, without loss of generality, that G is strongly connected, in which case subgraph $C(G)$ will also be strongly connected (see the proof of Lemma 5.1 below). The certificate uses the concept of *divergent spanning trees* [Georgiadis and Tarjan 2015]. In this context, a spanning tree T of a flow graph $G(s)$ is a tree with root s that contains a path from s to v for all vertices v. Two spanning trees B and R rooted at s are *divergent* if for all vertices v, the paths from s to v in B and R share only the dominators of v. Every

[2]Of course, one can verify that v and w are not 2-edge-connected in $O(m)$ time without a witness, by applying two iterations of the Ford-Fulkerson method [Ford and Fulkerson 2010] in each direction. The witness, however, makes the verification trivial to implement.

flow graph $G(s)$ has two such spanning trees, computable in linear time [Georgiadis and Tarjan 2015]. Moreover, the computed spanning trees are*maximally edge-disjoint*, meaning that the only edges they have in common are the bridges of $G(s)$.

The sparse certificate can be constructed during the computation of the 2-edge-connected blocks by extending Algorithm Fast2ECB. We now sketch the main modifications needed. During the execution of Algorithm Fast2ECB, we maintain a list (multiset) L of the edges to be added in $C(G)$. The same edge may be inserted into L multiple times, but the total number of insertions will be $O(n)$. Then, we can use radix sort to remove duplicate edges in $O(n)$ time. We initialize L to be empty. During Step 1 of Algorithm Fast2ECB, we compute two divergent spanning trees, $B(G(s))$ and $R(G(s))$ of $G(s)$, and insert their edges into L. Next, in Step 3.1, we compute two divergent spanning trees $B(H^R(r))$ and $R(H^R(r))$ for each auxiliary graph $H^R(r)$. For each edge (u, v) of these spanning trees, we insert a corresponding edge into L as follows. If both u and v are ordinary vertices in $H^R(r)$, we insert (u, v) into L, since it is an original edge of G. Otherwise, u or v is an auxiliary vertex and we insert into L a corresponding original edge of G. Such an original edge can be easily found during the construction of the auxiliary graphs. Finally, in Step 3.4, we compute two spanning trees for every connected component S_i of each auxiliary graph $H_q^R \setminus (p, q)$ as follows. Let H_{S_i} be the subgraph of H_q that is induced by the vertices in S_i. We choose an arbitrary vertex $v \in S_i$ and compute a spanning tree of $H_{S_i}(v)$ and a spanning tree of $H_{S_i}^R(v)$. We insert in L the original edges that correspond to the edges of these spanning trees.

LEMMA 5.1. *The sparse certificate $C(G)$ has the same 2-edge-connected blocks as the input digraph G.*

PROOF. It suffices to show that the execution of Algorithm Fast2ECB on $C(G)$ produces the same 2-edge-connected blocks as the execution of Algorithm Fast2ECB on G. The correctness of Algorithm Fast2ECB implies that it produces the same result regardless of the choice of start vertex s. So, we assume that both executions choose the same start vertex s. We will refer to the execution of Algorithm Fast2ECB with input G (resp., $C(G)$) as Fast2ECB(G) (resp., Fast2ECB($C(G)$)).

First, we note that $C(G)$ is strongly connected. This follows from the fact that $C(G)$ contains a spanning tree of $G(s)$ and that it also contains edges that correspond to a spanning tree for the reverse of each auxiliary graph G_r; if (u, v) is a shortcut edge in such a spanning tree for the reverse auxiliary graph H^R, then $C(G)$ will contain original edges that form a path from v to u. Moreover, the fact that $C(G)$ contains two divergent spanning trees of G implies that G and $C(G)$ have the same dominator tree and bridges, with respect to the start vertex s that are computed in Step 1. Hence, the subtrees $T(r)$ computed in Step 2 of Algorithm Fast2ECB are the same in both executions Fast2ECB(G) and Fast2ECB($C(G)$). The same argument as in Step 1 implies that in Steps 3.1 and 3.2, both executions Fast2ECB(G) and Fast2ECB($C(G)$) compute the same partitions $T^R(r)$ of each auxiliary graph $H^R(r)$. Finally, by construction, the strongly connected components of each auxiliary graph $H_q^R \setminus (p, q)$ are the same in both executions of Fast2ECB(G) and Fast2ECB($C(G)$).

We conclude that Fast2ECB(G) and Fast2ECB($C(G)$) compute the same 2-edge-connected blocks as claimed. □

The fact that $C(G)$ has $O(n)$ edges follows from Lemmas 2.1 and 3.9. Therefore, we have the following result.

COROLLARY 5.2. *We can compute in linear time a sparse certificate for the 2-edge-connected blocks of a digraph.*

We note that our sparse certificate gives a linear-time constant approximation of the following optimization problem, which we refer to as the smallest 2-edge-connected blocks preserving spanning subgraph (2ECBS) problem, introduced by Jaberi [2015]. Given a strongly connected digraph G, we wish to compute the smallest strongly connected spanning subgraph of G that maintains the same 2-edge-connected blocks. This problem is NP-hard since it is related to two well-known NP-hard problems [Garey and Johnson 1979]: computing the smallest strongly connected spanning subgraph (SCSS) and computing the smallest 2-edge-connected spanning subgraph (2ECSS). Note that if all vertices in G are 2-edge-connected, then we have an instance of a 2-edge-connected spanning subgraph (2ECSS), while if there is no pair of 2-edge-connected vertices in G blocks, then we have an instance of a strongly connected spanning subgraph (SCSS). A careful analysis of a variant of our sparse certificate is given in Georgiadis et al. [2015b], where it is shown that it achieves a 4-approximation for 2ECBS.

6. CONCLUDING REMARKS AND OPEN PROBLEMS

In this article, we have studied 2-edge connectivity in directed graphs. In particular, we have presented a linear-time algorithm for the 2-*edge-connected* relation among the vertices of a digraph $G = (V, E)$, which defines a partition of V, the 2-*edge-connected blocks* of G. Once the 2-edge-connected blocks of G are available, it is straightforward to check in constant time if any two vertices are 2-edge-connected. We mention that we have implemented the algorithms described in this article and performed a thorough experimental evaluation on very large graphs (with millions of vertices and edges) [Di Luigi et al. 2015]; in those experiments, Algorithm Fast2ECB performed very well in practice.

In follow-up work [Georgiadis et al. 2015a], we showed how to adapt the techniques we developed in this article in order to obtain a linear-time algorithm for computing the 2-vertex-connected blocks of a digraph. In this case, some technical details become more complicated and the approach requires several new ideas. We leave as an open question if the 2-edge-connected or the 2-vertex-connected components of a digraph can be computed in linear time. The best current bound for both problems is $O(n^2)$ [Henzinger et al. 2015].

ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments.

REFERENCES

S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. 1999. Dominators in linear time. *SIAM J. Comput.* 28, 6 (1999), 2117–32.

J. Bang-Jensen and G. Gutin. 2002. *Digraphs: Theory, Algorithms and Applications (Springer Monographs in Mathematics)* (1st ed. 2001. 3rd printing ed.). Springer.

A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. 2008. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.* 38, 4 (2008), 1533–1573.

A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. 1998. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems* 20, 6 (1998), 1265–96. Corrigendum in 27(3):383-7, 2005.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. 1991. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2015. 2-connectivity in directed graphs: An experimental study. In *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments*. 173–187. DOI:http://dx.doi.org/10.1137/1.9781611973754.15

Ya. M. Erusalimskii and G. G. Svetlov. 1980. Bijoin points, bibridges, and biblocks of directed graphs. *Cybernetics* 16, 1 (1980), 41–44. DOI:http://dx.doi.org/10.1007/BF01099359

D. Firmani, G. F. Italiano, L. Laura, A. Orlandi, and F. Santaroni. 2012. Computing strong articulation points and strong bridges in large scale graphs. In *Proceedings of the 10th International Symposium on Experimental Algorithms*. 195–207.

D. R. Ford and D. R. Fulkerson. 2010. *Flows in Networks*. Princeton University Press, Princeton, NJ.

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

H. N. Gabow. 2016. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms* 12, 2, Article 24, 73 pages. DOI:http://dx.doi.org/10.1145/2764909

H. N. Gabow and R. E. Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.* 30, 2 (1985), 209–21.

M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY.

L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2015a. 2-vertex connectivity in directed graphs. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*. 605–616.

L. Georgiadis, G. F. Italiano, C. Papadopoulos, and N. Parotsidis. 2015b. Approximating the smallest spanning subgraph for 2-edge-connectivity in directed graphs. In *Proceedings of the 23rd European Symposium on Algorithms*. 582–594.

L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan. 2014. Loop nesting forests, dominators, and applications. In *Proceedings of the 13th International Symposium on Experimental Algorithms*. 174–186.

L. Georgiadis and R. E. Tarjan. 2004. Finding dominators revisited. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*. 862–871.

L. Georgiadis and R. E. Tarjan. 2015. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms* 12, 1, Article 11, 42 pages. DOI:http://dx.doi.org/10.1145/2764913

Y. Guo, F. Kuipers, and P. Van Mieghem. 2003. Link-disjoint paths for reliable QoS routing. *International Journal of Communication Systems* 16, 9 (2003), 779–798. DOI:http://dx.doi.org/10.1002/dac.612

M. Henzinger, S. Krinninger, and V. Loitzenbauer. 2015. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*. 713–724.

A. Itai and M. Rodeh. 1988. The multi-tree approach to reliability in distributed networks. *Information and Computation* 79, 1, 43–59.

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

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

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

T. Lengauer and R. E. Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems* 1, 1, 121–41.

K. Menger. 1927. Zur allgemeinen kurventheorie. *Fund. Math.* 10, 96–115.

H. Nagamochi and T. Ibaraki. 1992. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica* 7, 583–596.

H. Nagamochi and T. Ibaraki. 2008. *Algorithmic Aspects of Graph Connectivity*, 1st ed. Cambridge University Press.

H. Nagamochi and T. Watanabe. 1993. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E76A, 4, 513–517.

R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2, 146–160.

R. E. Tarjan. 1974a. *Edge-Disjoint Spanning Trees, Dominators, and Depth-First Search*. Technical Report. Stanford University, Stanford, CA.

R. E. Tarjan. 1974b. Finding dominators in directed graphs. *SIAM J. Comput.* 3, 1, 62–89.

R. E. Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM* 22, 2, 215–225.

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

Received September 2015; revised June 2016; accepted July 2016