2-Vertex Connectivity in Directed Graphs

Loukas Georgiadis\textsuperscript{1} Giuseppe F. Italiano\textsuperscript{2} Luigi Laura\textsuperscript{3} Nikos Parotsidis\textsuperscript{1}

September 23, 2014

Abstract

We complement our study of 2-connectivity in directed graphs \cite{7}, by considering the computation of the following 2-vertex-connectivity relations: We say that two vertices $v$ and $w$ are 2-vertex-connected if there are two internally vertex-disjoint paths from $v$ to $w$ and two internally vertex-disjoint paths from $w$ to $v$. We also say that $v$ and $w$ are vertex-resilient if the removal of any vertex different from $v$ and $w$ leaves $v$ and $w$ in the same strongly connected component. We show how to compute the above relations in linear time so that we can report in constant time if two vertices are 2-vertex-connected or if they are vertex-resilient. We also show how to compute in linear time a sparse certificate for these relations, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-vertex-connectivity and vertex-resilience relations as the input graph, where $n$ is the number of vertices.

1 Introduction

Let $G = (V,E)$ be a directed graph (digraph), with $m$ edges and $n$ vertices. A vertex (resp., an edge) of $G$ is a strong articulation point (resp., a strong bridge) if its removal increases the number of strongly connected components of $G$. A digraph $G$ is 2-vertex-connected if it has at least three vertices and no strong articulation points; $G$ is 2-edge-connected if it has no strong bridges. The 2-vertex-connected (resp., 2-edge-connected) components of $G$ are its maximal 2-vertex-connected (resp., 2-edge-connected) subgraphs.

Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications \cite{2, 15}. Although the concept of 2-vertex-connected and 2-edge-connected components is useful in many applications, it does not capture the pairwise connectivity among the vertices of a digraph in the following sense. Two vertices may lie in different 2-vertex-connected and 2-edge-connected components but still be connected by several disjoint paths. This observation motivates the following natural 2-vertex and 2-edge connectivity relations \cite{7, 11, 16}. Let $v$ and $w$ be two distinct vertices. We say that $v$ and $w$ are 2-vertex-connected (resp., 2-edge-connected), and we denote this relation by $v \leftrightarrow_2 w$ (resp., $v \leftrightarrow_{2e} w$), if there are two internally vertex-disjoint (resp., two edge-disjoint) directed paths from $v$ to $w$ and two internally vertex-disjoint (resp., two edge-disjoint) directed paths from $w$ 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.) We define a 2-vertex-connected block (resp., 2-edge-connected
Figure 1: Vertices \(a\) and \(f\) are vertex-resilient but not 2-vertex-connected because edge \((f, a)\) is a strong bridge. Vertices \(a\) and \(g\) are 2-vertex-connected.

block) of a digraph \(G = (V, E)\) as a maximal subset \(B \subseteq V\) such that \(u \leftrightarrow_{2v} v\) (resp., \(u \leftrightarrow_{2e} v\)) for all \(u, v \in B\). We say that \(v\) and \(w\) are vertex-resilient (resp., edge-resilient), denoted by \(v \leftrightarrow_{vr} w\) (resp., \(v \leftrightarrow_{er} w\)), if the removal of any vertex different from \(v\) and \(w\) (resp., any edge) leaves \(v\) and \(w\) in the same strongly connected component. We define a vertex-resilient block (resp., edge-resilient block) of a digraph \(G = (V, E)\) as a maximal subset \(B \subseteq V\) such that \(u \leftrightarrow_{vr} v\) (resp., \(u \leftrightarrow_{er} v\)) for all \(u, v \in B\). By Menger’s Theorem \([14]\), \(v \leftrightarrow_{2e} w\) if and only if \(v \leftrightarrow_{er} w\), so these two relations are identical. On the other hand, two vertices \(v\) and \(w\) that are vertex-resilient are not necessarily 2-vertex-connected. See Figure 1. If, however, \(v\) and \(w\) are not adjacent then \(v \leftrightarrow_{2v} w\) if and only if \(v \leftrightarrow_{vr} w\).

For undirected graphs, the 2-vertex-connected (resp., 2-edge-connected) blocks are identical to the 2-vertex-connected (resp., 2-edge-connected) components. Moreover, all bridges, articulation points, 2-edge- and 2-vertex-connected components of undirected graphs can be computed in linear time \([17]\). The corresponding problems on digraphs appear to be more difficult. Although all the strong bridges and strong articulation points of a digraph can be found in linear time \([10]\), the best current bound for computing the 2-edge- and the 2-vertex-connected components in a digraph is \(O(mn)\) \([12]\). In \([7]\) we presented a linear-time algorithm to compute the 2-edge-connected blocks of a digraph \(G = (V, E)\). Since the 2-edge-connected relation defines a partition of \(V\), given the 2-edge-connected blocks we can test in constant time if two vertices are 2-edge-connected. Here we consider the computation of the 2-vertex-connected blocks and the vertex-resilient blocks, and provide linear-time algorithms. Despite the fact that these blocks do not give a partition of \(V\), we show that we can test in constant time if any two vertices are 2-vertex-connected or vertex-resilient.

Independently from our work, Jaberi \([11]\) considered the computation of the 2-vertex-connected, 2-edge-connected, and vertex-resilient blocks. His algorithms, however, require \(O(mn)\) time in the worst case and \(O(n^2)\) space.

2 Flow graphs, dominators, and bridges

In this section we introduce some terminology that will be useful throughout the paper. 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 $w$.

Lengauer and Tarjan \cite{LengauerTarjan87} presented an algorithm for computing dominators in $O(m\alpha(n,m/n))$ time for a flow graph with $n$ vertices and $m$ edges, where $\alpha$ is a functional inverse of Ackermann’s function \cite{Ackermann50}. Subsequently, several linear-time algorithms were discovered \cite{BFP89,KS00,KS00a,KS01,KS01a}. Italiano et al. \cite{IKS93} 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 digraph that results from $G$ after reversing edge directions; similarly, the strong bridges of $G$ correspond to the bridges of $G(s)$ and $G^R(s)$.

Let $T$ be a rooted tree whose vertex set is $V$. Tree $T$ has the *parent property* if for all $(v, w) \in E$, the parent of $w$ is an ancestor of $v$ in $T$. Tree $T$ has the *siblings property* if $v$ does not dominate $w$ for all siblings $v$ and $w$. The parent and sibling properties are necessary and sufficient for a tree to be the dominator tree.

**Theorem 2.1.** \cite{Gusfield91} Tree $D$ has the parent and sibling properties.

## 3 Vertex-resilient blocks and 2-vertex-connected blocks

In this section we provide some basic properties of the vertex-resilient blocks and the 2-vertex-connected blocks. In particular, we show that any digraph has at most $n - 1$ vertex-resilient (resp., 2-vertex-connected) blocks and, moreover, that there is a forest representation of these blocks that enables us to test vertex-resilience (resp., 2-vertex-connectivity) between any two vertices in constant time. This structure is similar to the one used in \cite{Gusfield95} to represent the biconnected components of an undirected graph.

**Lemma 3.1.** Let $u$, $v$, $x$, and $y$ be distinct vertices such that $u \leftrightarrow_{\text{vr}} x$, $v \leftrightarrow_{\text{vr}} x$, $u \leftrightarrow_{\text{vr}} y$, and $v \leftrightarrow_{\text{vr}} y$. Then also $x \leftrightarrow_{\text{vr}} y$ and $u \leftrightarrow_{\text{vr}} v$.

**Proof.** Assume, for contradiction, that $x$ and $y$ are not vertex-resilient. Then there is a strong articulation point $w$ such that every path from $y$ to $x$ contains $w$, or every path from $x$ to $y$ contains $w$ (or both). Without loss of generality, suppose that $w$ is contained in every path from $y$ to $x$. Since $u$ and $v$ are distinct, we can assume that $w \neq u$. (If $w = u$ then we swap the role of $u$ and $v$.). Then, $y \leftrightarrow_{\text{vr}} u$ implies that there is a path $P$ from $y$ to $u$ that avoids $w$, and similarly, $u \leftrightarrow_{\text{vr}} x$ implies that there is a path $Q$ from $u$ to $x$ that avoids $w$. So, $P$ followed by $Q$ gives a path from $y$ to $x$ that does not contain $w$, a contradiction. Hence $x \leftrightarrow_{\text{vr}} y$. The fact that $u \leftrightarrow_{\text{vr}} v$ follows by repeating the same argument for $u$ and $v$. \hfill $\square$

**Corollary 3.2.** Let $B$ and $B'$ be two distinct vertex-resilient blocks of a digraph $G = (V, E)$. Then $|B \cap B'| \leq 1$.

**Proof.** Follows immediately from Lemma 3.1. \hfill $\square$

We denote by $\text{VRB}(u)$ the vertex-resilient blocks that contain $u$. Define the *block graph* $F = (V_F, E_F)$ of $G$ as follows. The vertex set $V_F$ consists of the vertices in $V$ and also contains one *block node* for each vertex-resilient block of $G$. The edge set $E_F$ consists of the edges $\{u, B\}$ where $B \in \text{VRB}(u)$. Thus, $F$ is an undirected bipartite graph. Next we show that it is also acyclic.
Lemma 3.3. Let $u$ and $v$ be any vertices that are connected by a path $P$ in $F$. Then, for any vertex $w$ not on $P$, $u$ and $v$ are in the same strongly connected component in digraph $G \setminus w$.

Proof. It suffices to show that $G$ contains a path $Q$ from $u$ to $v$ that avoids $w$. The same argument shows that $G$ contains a path from $v$ to $u$ that avoids $w$. Let $P = (u_1 = u, B_1, v_2, B_2, \ldots, u_{k+1} = v)$. Then $u_i \leftrightarrow_{v_i} u_{i+1}$, for $1 \leq i \leq k$, so there is a path $P_i$ in $G$ from $v_i$ to $v_{i+1}$ that avoids $w$. Then the catenation of paths $P_1, \ldots, P_k$ gives a path in $G$ from $u$ to $v$ that avoids $w$. □

Lemma 3.4. Graph $F$ is acyclic.

Proof. Suppose, for contradiction, that $F$ contains a cycle $C$. We show that all vertices $w \in C \cap V$ belong to the same vertex-resilient block $B$. Let $u, v \in V$ be two vertices on a minimal cycle $C$ of $F$ that are adjacent to a block node $B$. (Such $u, v,$ and $B$ exist since $F$ is bipartite.) Then, $u$ and $v$ cannot be the only vertices in $V$ that are on $C$, since otherwise they would be adjacent to another block $B'$ on $C$, violating Corollary 3.2. Therefore, $C$ contains a vertex $w \in V \setminus \{u, v\}$. Clearly, $w \notin B$, otherwise the edge $\{w, B\}$ would exist contradicting the minimality of $C$. Hence, there is a vertex $z \in B$ such that all paths from $z$ to $w$ contain a common strong articulation point or all paths from $w$ to $z$ contain a common strong articulation point. Suppose, without loss of generality, that a vertex $x$ is contained in every path from $z$ to $w$. Let $P$ be the path that results from $C$ by removing $B$. Let $P_u$ and $P_v$ be the subpaths of $P$ from $u$ to $w$ and from $v$ to $w$, respectively. Then $x \notin P_u$ or $x \notin P_v$ (or both). Suppose $x \notin P_u$; if not then swap the role of $u$ and $v$. Then, by Lemma 3.3 there is a path $Q$ in $G$ from $u$ to $w$ that avoids $x$. Also, since $u \leftrightarrow_{vt} z$, there is a path $Q'$ in $G$ from $z$ to $w$ that avoids $x$. Then the catenation of $Q'$ and $Q$ gives a path in $G$ from $z$ to $w$ that avoids $x$, a contradiction. □

Lemma 3.5. The number of vertex-resilient blocks in a digraph $G$ is at most $n - 1$.

Proof. We prove the lemma by showing that forest $F$ contains at most $n - 1$ block nodes. Since $F$ is a forest we can root each tree $T$ of $F$ at some arbitrary vertex $r$. Every level of $T$ contains either only vertices of $V$ or only block nodes, because $F$ is bipartite. Moreover, every block node is adjacent to at least two vertices of $V$, due to the fact that each vertex-resilient block in $G$ contains at least 2 vertices. Hence, every leaf of $T$ is a vertex in $V$. Now consider a partition of $T$ into vertex disjoint paths $P_1, P_2, \ldots, P_k$, such that each $P_i$ leads from some vertex or block node to a leaf descendant. The number of block nodes in each $P_i$ is at most equal to $|P_i \cap V|$. Also, in the path $P_i$ starting at $r$ the number of block nodes in $P_i$ is less than $|P_i \cap V|$. We conclude that there at most $n - 1$ block nodes in $F$. □

Lemma 3.6. The total number of vertices in all vertex-resilient blocks is at most $2n - 2$.

Proof. By Lemmas 3.4 and 3.5, the block graph $F$ is a forest with at most $2n - 1$ vertices. Each occurrence of a vertex $v$ in a block $B$ corresponds to an edge $\{v, B\}$ of $F$. Therefore, the total number of vertices in all vertex-resilient blocks equals the number of edges in $F$, and the lemma follows. □

We turn $F$ into a forest of rooted trees, by choosing an arbitrary vertex as the root of each tree. Then $u \leftrightarrow_{vt} w$ if and only if $u$ and $w$ are siblings or one the grandparent of the other. See Figure 2. We can perform both tests in constant time simply by storing the parent of each vertex in $F$. Thus, we can test in constant time if two vertices are vertex-resilient.

Now we turn to 2-vertex-connected blocks. As we already mentioned in the introduction, Menger’s Theorem [1] implies that if $v$ and $w$ are not adjacent then $v \leftrightarrow_{2v} w$ if and only if $v \leftrightarrow_{vt} w$. If, on the other hand, $v \leftrightarrow_{vt} w$ but $v$ and $w$ are not 2-vertex-connected, then at least
one of the edges \((v, w)\) and \((w, v)\) exists in \(G\) and is a strong bridge. This observation implies the following statement that relates 2-vertex-connected, 2-edge-connected and vertex-resilient blocks.

**Lemma 3.7.** For any two distinct vertices \(v\) and \(w\), \(v \leftrightarrow_{2v} w\) if and only if \(v \leftrightarrow_{v1} w\) and \(v \leftrightarrow_{2e} w\).

By Lemma 3.7, we have that the 2-vertex-connected blocks are refinements of the vertex-resilient blocks, formed by the intersections of the vertex-resilient blocks and the 2-edge-connected blocks of the digraph \(G\). Since the 2-edge-connected blocks are a partition of the vertices of \(G\), these intersections partition each vertex-resilient block. From this property we conclude that Lemmas 3.1, 3.2, 3.4, and 3.5 also hold for the 2-vertex-connected blocks.

### 4 Computing the vertex-resilient blocks

In this section we develop algorithms that compute the vertex-resilient blocks of a digraph \(G\). We can assume that \(G\) is strongly connected, so \(m \geq n\). If not, then we process each strongly connected component separately; if \(u \leftrightarrow_{v1} v\) 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 articulation point at a time. In order to get a more efficient solution, we need to consider simultaneously how different strong articulation points divide the vertices into blocks, which we do with the help of dominator trees. We achieve linear running time by combining the simple algorithm with the dominator-tree-based division.

#### 4.1 A simple algorithm

Algorithm SimpleVRB is an immediate application of the characterization of the vertex-resilient blocks in terms of strong articulation points. Let \(u\) and \(v\) be two distinct vertices. We say that a strong articulation point \(x\) separates \(u\) from \(v\) if all paths from \(u\) to \(v\) contain \(x\). In this case \(u\) and \(v\) belong to different strongly connected components of \(G \setminus x\). This observation implies that we can compute the vertex-resilient blocks by computing the strongly connected components of \(G \setminus x\) for every strong articulation point \(x\).
Algorithm SimpleVRB: Computation of the vertex-resilient blocks of a strongly connected digraph $G = (V, E)$

**Step 1:** Compute the strong articulation points of $G$.

**Step 2:** Initialize the current set of blocks as $B = \{V\}$. (Start from the trivial set containing only one block.)

**Step 3:** For each strong articulation point $x$ do:

- **Step 3.1:** Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus x$. Let $S$ be the partition of $V$ defined by the strongly connected components $S_i$.
- **Step 3.2:** Execute $\text{refine}(B, S, x)$.

To do this efficiently we define an operation that refines the currently computed blocks. Let $B$ be a set of blocks, let $S$ be a partition of a set $U \subseteq V$, and let $x$ be a vertex not in $U$.

$\text{refine}(B, S, x)$: For each block $B \in B$, substitute $B$ by the sets $B \cap (S \cup \{x\})$ of size at least two, for all $S \in S$.

**Lemma 4.1.** Let $N$ be the total number of elements in all sets of $B$ ($N = \sum_{B \in B} |B|$), and let $K$ be the number of elements in $U$. Then, the operation $\text{refine}(B, S, x)$ can be executed in $O(N + K)$ time.

**Proof.** A simple way to achieve the claimed bound is to number the sets of the partition $S$, each with a distinct integer id in the interval $[1, K]$. Consider a block $B$. Each element $v \in B$ is assigned a label that is equal to id of the set $S \in S$ that contains $v$ if $v \in U$, and zero otherwise. Then, the computation of the sets $B \cap (S \cup \{x\})$ for all $S \in S$ can be done in $O(|B|)$ time with bucket sorting.

**Lemma 4.2.** Algorithm SimpleVRB runs in $O(mp^*)$ time, where $p^*$ is the number of strong articulation points of $G$.

**Proof.** The strong articulation points of $G$ can be computed in linear time by \[10\]. In each iteration of Step 3, we can compute the strongly connected components of $G \setminus x$ in linear time \[17\]. As we discover the $i$-th strongly connected component, we assign label $i$ ($i \in \{1, \ldots, n\}$) to the vertices in $S_i$. By Lemma \[3.5\], the number of vertex-resilient blocks of $G$ is at most $n - 1$. Therefore, since the number of blocks does cannot decrease during any iteration, $B$ contains at most $n - 1$ blocks in each execution of Step 3. By induction on the number of iterations, it follows that the algorithm maintains the invariant that any two distinct blocks in $B$ have at most one element in common, and that the corresponding block graph is a forest. Therefore, by Lemma \[3.5\], the total number of elements in all blocks is at most $2n - 2$. So, by Lemma \[4.1\] each iteration of Step 3 takes $O(n)$ time.

We note that a digraph may have up to $n - 1$ strong articulation points, so the above running time is $\Theta(mn)$ in the worst case.

In Section \[5\] we apply the $\text{refine}$ operation in order to compute the 2-vertex-connected blocks from the vertex-resilient blocks and the 2-edge-connected blocks. To that end, we define a version of this operation, $\text{refine}(B, S)$, which corresponds to $\text{refine}(B, S, x)$ with $x = \text{null}$.
4.2 Linear-time algorithm

We will show how to obtain a faster algorithm by applying the framework developed in [7] for the computation of the 2-edge-connected blocks. That is, we will use dominator trees and auxiliary graphs. The latter, however, are defined in a substantially different way which complicates several technical details. Another complication arises from the more technical details. Another complication arises from the more

As a warm up, first consider the computation of VRB(v), i.e., of the vertex-resilient blocks that contain a specific vertex v. Consider the flow graph G(v) with root v and its reverse G^R(v). Let w be a vertex other than v.

Clearly, v and w are vertex-resilient if and only if v is the only proper dominator of w in both G(v) and G^R(v), i.e., d(w) = v and d^R(w) = v. Now let u be a sibling of w in both D(v) and D^R(v). The fact that d^R(w) = v and d(u) = v implies that for any vertex x ∈ V \ {v, w, u} there is path from w to u through v that avoids x. So w and u are in a common vertex-resilient block that contains v if and only if they lie in the same strongly connected component of G \ v. This observation implies the following linear-time algorithm to compute the vertex-resilient blocks that contain v. Compute the dominator trees D(v) and D^R(v) of G(v) and G^R(v) respectively. Let C(v) (resp., C^R(v)) be the set of children of v in G(v) (resp., G^R(v)). Set U = C(v) ∩ C^R(v) and initialize the set of blocks B = {U}. Compute the strongly connected blocks S_1, S_2, ..., S_k of G \ v. Let S be the set that contains the nonempty restrictions of the S_i sets to U, i.e., S contains the nonempty sets S_i ∩ U. Finally, execute refine(B, S, v).

Note that all the vertex-resilient blocks can be computed in O(mn) time by applying the above algorithm to all vertices v. To avoid the repeated applications of this algorithm we use the concept of auxiliary graphs from [7]. In our case, however, we need a substantially different definition of auxiliary graphs that we give next. Before we do that, we state two properties regarding information that a dominator tree can provide about vertex-resilient blocks and paths.

Lemma 4.3. Let G = (V, E) be a strongly connected graph, and let s ∈ V be an arbitrary start vertex. Any two vertices x and y are vertex-resilient only if they are siblings in D(s) or one is the immediate dominator of the other in G(s).

Proof. Immediate. □

Lemma 4.4. Let r be a vertex, and 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 proper descendants of r in D(s). Moreover, all simple paths from v to any descendant of r in D(s) contain r.

Proof. Let P be any path from v to r. (Such a path exists since graph G is strongly connected.) Let u be the first vertex on P such that u is a descendant of r. Then either u = 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 u is a path from s to u that avoids r, a contradiction. □

4.2.1 Auxiliary graphs

Let s be an arbitrarily chosen start vertex in G. Recall that we denote by G(s) the flow graph with start vertex s, by G^R(s) the flow graph obtained from G(s) after reversing edge directions, by D(s) and D^R(s) the dominator trees of G(s) and G^R(s) respectively, and by C(v) and C^R(v) the set of children of v in D(s) and D^R(s) respectively.

As in [7], auxiliary graphs are a key concept in our algorithm that provides a decomposition of the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the
original vertex-resilient blocks. Here, however, we need a more complicated definition of auxiliary graphs.

For each vertex \( r \neq s \) that is not a leaf in \( D(s) \) we build the auxiliary graph \( G_r = (V_r, E_r) \) of \( r \) as follows. Let \( C^k(r) \) denote the level \( k \) descendants of \( r \), i.e., \( C^0(r) = \{ r \} \), \( C^1(r) = C(r) \), etc. The vertex set of \( G_r \) is \( V_r = \bigcup_{k=0}^3 C^k(r) \) and it is partitioned into a set of ordinary vertices \( V_r^o = C^1(r) \cup C^2(r) \) and a set of auxiliary vertices \( V_r^a = C^0(r) \cup C^3(r) \). The edge set \( E_r \) contains all edges in \( G = (V, E) \) induced by the vertices in \( V_r \) (i.e., edges \((u, v) \in E \) such that \( u \in V_r \) and \( v \in V_r \)). We also add in \( E_r \) the following types of shortcut edges that correspond to paths in \( G_r \).

(a) If \( G \) contains an edge \((u, v)\) such that \( u \notin V_r \) is a descendant of \( r \) in \( D(s) \) and \( v \in V_r \) then we add the shortcut edge \((z, v)\) where \( z \) the is an ancestor of \( u \) in \( D(s) \) such that \( z \in C^3(r) \). (b) If \( G \) contains an edge \((u, v)\) such that \( u \) but not \( v \) is a descendant of \( r \) in \( D(s) \) then we add the shortcut edge \((z, r)\) where \( z \) the nearest ancestor of \( u \) in \( D(s) \) such that \( z \in V_r \) \((z = u \text{ if } u \in V_r)\). We note that we do not keep multiple (parallel) shortcut edges. See Figure 3. We use the same definition for the auxiliary graph \( G_s \) of \( s \), with the only difference that we let \( s \) be an ordinary vertex. Also note that \( G_s \) does not contain type-(b) shortcut edges.

**Lemma 4.5.** Let \( v \) and \( w \) be two vertices in \( V_r \). Any path from \( v \) to \( w \) in \( G \) has a corresponding path from \( v \) to \( w \) in \( G_r \), and vice versa.

*Proof.* Consider a path \( P \) from \( v \) to \( w \) in \( G \). We show that it has a corresponding path \( P_r \) from \( v \) to \( w \) in \( G_r \). If \( P \) consists only of vertices in \( G_r \) then \( P_r = P \). Otherwise, let \((u, x)\) be the first edge on \( P \) such that \( u \in V_r \) and \( x \notin V_r \). Also let \((y, z)\) be the first edge on \( P \) after \((u, x)\) such that \( y \notin V_r \) and \( z \in V_r \). Theorem 2.4 implies: (i) For the edge \((u, x)\) we have that either \( u = d(x) \) or \( x \) is not an descendant of \( r \) in \( D(s) \) and \( d(x) \) is a proper ancestor of \( r \). (ii) For the edge \((y, z)\) we have that either \( y \) is not a descendant of \( r \) and \( z = r \), or \( y \) is a descendant of a vertex in \( C^3(r) \).

Suppose \( u = d(x) \). Let \( t \) be the first vertex on \( P \) after \( x \) that is not an descendant of \( x \). If \( t \in V_r \) then \( t = z \). In this case the part of \( P \) from \( u \) to \( z \) corresponds to the type-(a) edge \((u, z)\) in \( P_r \). If \( t \notin V_r \) then \( t \) is not a descendant of \( r \) in \( D(s) \), and by Lemma 4.4 we have that \( z = r \). So the part of \( P \) from \( u \) to \( z \) corresponds to the type-(b) edge \((u, r)\) in \( P_r \). Now suppose that \( x \) is not a proper descendant of \( r \) in \( D(s) \). Then Lemma 4.4 implies that \( z = r \) so the part of \( P \) from \( u \) to \( z = r \) corresponds to the edge \((u, r)\) in \( P_r \). We can repeat the same argument for every part of \( P \) that is outside \( V_r \), which gives a valid path \( P_r \) in \( G_r \).

Now we prove that any path \( P_r \) from \( v \) to \( w \) in \( G_r \) has a corresponding path \( P \) from \( v \) to \( w \) in \( G \). It is sufficient to show that every edge \( e = (x, y) \) on \( P_r \) that is not an edge of \( G \) has a corresponding path \( P_e \) from \( x \) to \( y \) in \( G \). In this case, \( e \) is a type-(a) or a type-(b) shortcut edge in \( G_r \). Suppose \( e \) is of type (a). By the construction of \( G_r \), \( G \) has an edge \((z, y)\) where \( z \) is a descendant of \( x \). Then \( G \) contains a path \( Q \) from \( x \) to \( z \), so we have \( P_e = Q \cdot (z, y) \). Now suppose that \( e \) is of type (b). By the construction of \( G_r \), \( y = r \) and \( x \) is a descendant of \( r \) in \( D(s) \). Let \((z, t)\) be an edge of \( G \) that corresponds to \( e \). Then \( z \) is the nearest ancestor of \( x \) in \( V_r \). Let \( Q \) be a path from \( z \) to \( x \) in \( G \). By Lemma 4.4, there is a path \( Q' \) in \( G \) from \( y \) to \( r \) that does not contain any proper descendant of \( r \). Then \( P_e = Q \cdot Q' \).

**Corollary 4.6.** Each auxiliary graph \( G_r \) is strongly connected.

*Proof.* Follows from the construction of \( G_r \), Lemma 4.5 and the fact that \( G \) is strongly connected.

The next lemma shows that auxiliary graphs maintain the vertex-resilient relation of the original digraph.
Figure 3: A strongly connected graph $G$, the dominator tree $D(s)$ of flow graph $G(s)$, the auxiliary graph $H = G_r$ and the dominator tree $D^R_H(r)$ of the flow graph $H^R(r)$. (The edges of the dominator tree $D^R_H(r)$ are shown directed from child to parent.) The auxiliary vertices of $H$ are shown gray.

**Lemma 4.7.** Let $v$ and $w$ be any two distinct vertices of $G$. Then $v$ and $w$ are vertex-resilient in $G$ if and only if they are both ordinary vertices in an auxiliary graph $G_r$ and they are vertex-resilient in $G_r$.

**Proof.** Suppose first that $v$ or $w$ is $s$. Without loss of generality assume $v = s$. Then by Lemma 4.3 we have that $w \in C^1(r)$, so $v$ and $w$ are both ordinary vertices of $G_r$. Now consider that $v, w \in V \setminus s$. From Lemma 4.3 we have that $v$ and $w$ belong in a set $C^1(r) \cup C^2(r)$ so they are both ordinary vertices of $G_r$. Clearly if all paths from $v$ to $w$ in $G_r$ contain a common vertex (strong articulation point), then so do all paths from $v$ to $w$ in $G$ by Lemma 4.5. Now we prove the converse. Suppose all paths from $v$ to $w$ in $G$ contain a common vertex $u$. If $u \in V_r$ then also all paths from $v$ to $w$ in $G_r$ contain $u$. So suppose $u \not\in V_r$. Then $v$ is not an ancestor of $w$ in $D(s)$, since otherwise there would be a path from $v$ to $w$ that avoids $u$.

First consider that $u$ is a (proper) descendant of $r$ in $D(s)$. Since $v$ is not an ancestor of $w$ in
D(s), there is a vertex \( x \in C^3(r) \) that is an ancestor of \( u \). By Lemma 4.4, all paths from \( v \) to \( u \) in \( G \), and thus all paths from \( v \) to \( w \), contain \( x \). By Lemma 4.5, this is also true for all paths from \( v \) to \( w \) in \( G_r \).

Finally, if \( u \) is not a descendant of \( r \), Lemma 4.4 implies that all paths from \( u \) to \( w \) in \( G \) contain vertex \( r \). Hence, all paths from \( v \) to \( w \) in \( G \) contain \( r \), and so do all paths from \( v \) to \( w \) in \( G_r \) by Lemma 4.5.

Lemma 4.8. The auxiliary graphs \( G_r \) have at most \( 4n \) vertices and \( 4m + n \) edges in total.

Proof. A vertex of \( G \) may appear in at most four auxiliary graphs. Therefore, the total number of edges in all auxiliary graphs excluding type-(b) shortcut edges \((u,v)\) with \( u \not\in V_r \) is at most \( 4m \). A type-(b) shortcut edge \((u,v)\) with \( u \not\in V_r \) of \( G_r \) corresponds to a unique vertex in \( C^3(r) \), so there are at most \( n \) such edges.

To construct the auxiliary graphs \( G_r = (V_r, E_r) \) we need to specify how to compute the shortcut edges of type (a) and (b). To do this efficiently we need to test ancestor-descendant relations in \( D(s) \). There are several simple \( O(1) \)-time tests of this relation [13]. The most convenient one for us is to number the vertices of \( D(s) \) from 1 to \( n \) in preorder, and to compute the number of descendants of each vertex \( v \), which we denote by \( \text{size}(v) \). Then \( v \) is a descendant of \( r \) if and only if \( \text{pre}(r) \leq \text{pre}(v) < \text{pre}(r) + \text{size}(r) \).

Suppose \((u,v)\) is an edge of type (a). We need to find the ancestor \( z \) of \( u \) in \( D(s) \) such that \( z \in C^3(r) \). We process all such arcs of \( G_r \) as follows. We create a list \( B_r \) that contains the edges \((u,v)\) of type (a), and sort \( B_r \) in increasing preorder of \( u \). We create a second list \( B_r' \) that contains the vertices in \( C^3(r) \), and sort \( B_r \) in increasing preorder. Then, the shortcut edge of \((u,v)\) is \((z,v)\), where \( z \) is the last vertex in the sorted list \( B_r' \) such that \( \text{pre}(z) \leq \text{pre}(u) \). Thus the shortcut edges of type (a) can be computed in linear time by bucket sorting and merging. Now we consider the edges of type (b). For each vertex \( w \in C^3(r) \) we need to test if there is an edge \((u,v)\) in \( G \) such that \( u \) is a proper descendant of \( w \) and \( v \) is not a descendant of \( r \) in \( D(s) \). In this case, we add in \( G_r \) the edge \((w,r)\). To do this test efficiently, we assign to each edge \((u,v)\) a tag \( t(u,v) \) which we set equal to the preorder number of the nearest common ancestor of \( u \) and \( v \) in \( D(s) \). We can do this easily by using the parent property and the \( O(1) \)-time test of the ancestor-descendant relation as follows: \( t(u,v) = \text{pre}(u) \) if \( u \) is an ancestor of \( v \) in \( D(s) \), \( t(u,v) = \text{pre}(v) \) if \( v \) is an ancestor of \( u \) in \( D(s) \), and \( t(u,v) = \text{pre}(d(v)) \) otherwise. At each node \( w \neq s \) in \( D(s) \) we store a label \( \ell(w) \) which is the minimum tag of among the type-(b) edges \((w,v)\). Using these labels we compute for each \( w \neq s \) in \( D(s) \) the values \( \text{low}(w) = \min \{ \ell(v) \mid v \text{ is a descendant of } w \text{ in } D(s) \} \). These computations can be done in \( O(m) \) time by processing the tree \( D(s) \) in a bottom-up order. Now consider the auxiliary graph \( G_r \). We process the vertices in \( C^3(r) \). For each such vertex \( w \) we add the shortcut edge \((w,r)\) if \( \text{low}(z) < \text{pre}(r) \).

Lemma 4.9. We can compute all auxiliary graphs \( G_r \) in \( O(m) \) time.

4.3 Algorithm

Now we are ready to describe our linear-time algorithm FastVRB. It uses two levels of auxiliary graphs and applies one iteration of Algorithm SimpleVRB for each auxiliary graph of the second level. The algorithm uses different dominator trees, and applies Lemma 4.3 in order to identify the vertex-resilient blocks. Since different dominator trees may define different blocks (which by Lemma 4.3 are supersets of the vertex-resilient blocks), we will use an operation that we call split to combine the different blocks.
We begin by computing the dominator tree $D(s)$ for an arbitrary start vertex $s$. For any vertex $v$, we let $\hat{C}(v)$ denote the set containing $v$ and the children of $v$ in $D(s)$, i.e., $\hat{C}(v) = C(v) \cup \{v\}$. Lemma 4.3 gives an initial division of the vertices into blocks that are supersets of the vertex-resilient blocks. Specifically, the vertex-resilient blocks that contain $v$ are subsets of $\hat{C}(v)$ or $\hat{C}(d(v))$ (for $v \neq s$).

During the course of the algorithm, each vertex $v$ becomes associated with a set of blocks $B(v)$ that contain $v$, which are subsets of $\hat{C}(v)$ and $\hat{C}(d(v))$ if $v \neq s$. The blocks are refined by applying the refine operation of Section 4.11 and operation split that we define next, and at the end of the algorithm each set of blocks $B(v)$ will be equal to $\text{VRB}(v)$.

Let $B$ be a block and $T$ be a tree with vertex set $V(T) \supseteq B$. For any vertex $v \in V(T)$, let $\hat{C}_T(v)$ be the set containing $v$ and the children of $v$ in $T$.

**Lemma 4.10.** Let $N$ be the number of vertices in $V(T)$. Then, the operation split($B$, $T$) can be executed in $O(N)$ time.

**Proof.** We number the vertices of $T$ in preorder. Let $\text{pre}(v)$ be the preorder number of $v \in V(T)$. Let $t(v)$ be the parent of $v \neq r$ in $T$, where $r$ is the root of $T$. We associate each vertex $v \neq r$ in $B$ with two labels $\ell_1(v) = \text{pre}(t(v))$ and $\ell_2(v) = \text{pre}(v)$, and create two corresponding pairs $\langle \ell_1(v), v \rangle$. Also, if $r \in B$ we associate $r$ with one label $\ell_2(r) = \text{pre}(r)$, and create a corresponding pair $\langle \ell_2(r), r \rangle$. Each block created by the split operation consists of a set of at least two vertices $v \in B$ that are associated with a specific label. We can find these blocks by sorting the pairs $\langle \ell_1(v), v \rangle$ by label, which can be done in $O(N)$ time with bucket sort.

In a high level, the algorithm begins with a “coarse” block tree, induced by the $\hat{C}(v)$ sets of $D(s)$, which is then refined by the blocks defined from the dominator trees of the auxiliary graphs. The final vertex-resilient block forest is then computed by considering the strongly connected components of the second level auxiliary graphs, after removing their designated start vertex. The algorithms needs to keep track of the blocks that contain a specific vertex, and, conversely, of the vertices that are contained in a specific block. To facilitate this search we explicitly store the adjacency lists of the current block forest $F$. Recall that $F$ is bipartite, so the adjacency list of a vertex $v$ stores the blocks that contain $v$, and the adjacency list of a block node $B$ stores the vertices in $B$. Initially $F$ contains one block for each set $\hat{C}(v)$, for all vertices $v$ that are not leaves in $D(s)$. These blocks are later refined by executing the split and refine operations, which maintain the invariant that $F$ is a forest, and that any two distinct blocks have at most two vertices in common. When we execute a split or a refine operation we can update the adjacency lists of $F$, while maintaining the bounds given in Lemmas 4.1 and 4.10. Also, since during the execution of the algorithm the number of blocks can only increase, $F$ contains at most $n - 1$ blocks at any given time. This fact implies that Lemma 3.5 holds, so the total number of vertices and edges in $F$ is $O(n)$.

**Lemma 4.11.** Algorithm FastVCR is correct.

**Proof.** Let $u$ and $v$ be any vertices. If $u$ and $v$ are vertex-resilient in $G$, then by Lemma 4.7 they are vertex-resilient in both auxiliary graphs of $G$ and $G_r$ that contain them as ordinary vertices. This implies that the algorithm will correctly include them in the same block in Step 1 and will not separate them in Steps 3.3 and 3.5. So suppose that $u$ and $v$ are not vertex-resilient. Then, without loss of generality, we can assume that all paths from $u$ to $v$ contain a common strong articulation
Algorithm FastVRB: Linear-time computation of the vertex-resilient 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 $D(s)$. For any vertex $v$, let $C(v)$ be the set containing $v$ and the children of $v$ in $D(s)$. For every vertex $v$ that is not a leaf in $D(s)$, associate block $\bar{C}(v)$ with every vertex $w \in \bar{C}(v)$.

**Step 2:** Compute the auxiliary graphs $G_r$ for all vertices $r$ that are not leaves in $D(s)$.

**Step 3:** Process the vertices of $D(s)$ in bottom-up order. For each auxiliary graph $H = G_r$ with $r$ not a leaf in $D(s)$ do:

**Step 3.1:** Compute the dominator tree $T = D_H^R(r)$.

**Step 3.2:** Compute the set $B$ of blocks that contain vertices in $C(r)$.

**Step 3.3:** For each block $B \in B$ execute $\text{split}(B,T)$.

**Step 3.4:** Compute the auxiliary graphs $H_q^R$ for all vertices $q$ that are not leaves in $T$.

**Step 3.5:** For each auxiliary graph $H_q^R$ with $q$ not a leaf do:

**Step 3.5.1:** Compute the set $B_q$ of blocks that contain at least two ordinary vertices in $H_q^R$.

**Step 3.5.2:** Compute the set $S$ of the strongly connected components of $H_q^R \setminus q$.

**Step 3.5.3:** Refine the blocks in $B_q$ by executing $\text{refine}(B_q,S,q)$.

Thus, $d(v) \neq u$. We argue that all the blocks that contain $u$ and all the blocks that contain $v$ will be separated in some step of the algorithm.

First we observe that $u$ and $v$ can appear together in at most one of the blocks constructed in Step 1. Also, $u$ and $v$ can remain in at most one block after each $\text{split}$ operation ($u$ and $v$ cannot have at most one identical label $\ell_i(u) = \ell_j(v)$). So suppose that $u$ and $v$ are still contained in one common block just before the execution of Step 3.5. We will show that $u$ and $v$ will be separated after the $\text{refine}$ operation executed in Step 3.5.3. Since $u$ and $v$ were not separated by a $\text{split}$ operation, they are either siblings or one is the parent of the other in $D_H^R(r)$. Also, since $d(v) \neq u$ we have the following cases.

(a) $d(u) = v$. Then $u$ and $v$ are both ordinary vertices of the auxiliary graph $H = G_r$ with $r = d(v)$. Lemma 4.7 implies that $G_r$ contains a strong articulation point $x$ that separates $u$ from $v$. We argue that $x$ is a proper ancestor of $u$ in $D_H^R(r)$. If not, then $H^R$ contains a path $P^R$ from $u$ to $r$ that avoids $x$. Since $d(v) = r$, $H$ contains a path $Q$ from $r$ to $v$ that avoids $x$. Thus $P \cdot Q$ is a path in $H$ from $u$ to $v$ that avoids $x$, a contradiction. Now we claim that $q = d_H^R(u)$ is also a strong articulation point that separates $u$ from $v$. Suppose the claim is false. Then $x \neq q$, so $x$ is a proper ancestor of $q$ in $D_H^R(r)$. Let $P$ be a path from $u$ to $v$ that avoids $q$. Then $x$ is on $P$ since $x$ separates $u$ from $v$. Let $P_x$ be the part of $P$ from $u$ to $x$. Also, since $x$ is a proper ancestor of $q$ in $D_H^R(r)$, $H^R$ has a path $Q^R$ from $r$ to $x$ that avoids $q$. Then $P \cdot Q$ is a path in $H$ from $u$ to $r$ that avoids $q$, a contradiction. The claim implies that $u$ and $v$ are located in different strongly connected components of $H_q^R \setminus q$, so they are contained in different blocks computed in Step 3.5.3.

(b) $d(v) = d(u) = r$. Then $u$ and $v$ are both ordinary vertices of the auxiliary graph $H = G_r$. Lemma 4.7 implies that $G_r$ contains a strong articulation point $x$ that separates $u$ from $v$. By the same arguments as in case (a), it follows that $q = d_H^R(u)$ is a strong articulation point that
Lemma 4.12. Algorithm FastVRB runs in \( O(m) \) time.

Proof. We account for the total time spent on each step that Algorithm FastVRB executes. Step 1 takes \( O(m) \) time by \[3\], and Step 2 takes \( O(m) \) time by Lemma 4.9. From Lemma 4.8 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. Then, again by Lemma 4.8 the total size (number of vertices and edges) of all auxiliary graphs \( H^R \) for all \( H \), computed in Step 3.4, is still \( O(m) \) and they are also computed in \( O(m) \) total time by Lemma 4.9. Now consider the split operations. All these operations that occur during Step 3.3 for a specific auxiliary graph \( G_r \) operate on the same tree \( T \), which can be preprocessed once, as in Lemma 4.10, for all split operations. Therefore, the total preprocessing time for all split operations is \( O(n) \). Excluding the preprocessing time for \( T \), a \( \text{split}(B,T) \) operation takes time proportional to the number of vertices in \( B \). Therefore all split operations take \( O(n) \) time in total by Lemmas 3.6 and 4.10. In Step 3.5.1 we examine the adjacency lists of the ordinary vertices \( v \in H^R \) and find the corresponding blocks that contain at least such two ordinary vertices. Then we examine the adjacency lists of each such block. So, the adjacency lists of each vertex \( v \) and each block that contains \( v \) can be examined at most three times. Hence, Step 3.5.1 takes \( O(n) \) time in total. Finally, Steps 3.5.2 and 3.5.3 take \( O(m) \) time in total by \[17\] and Lemmas 3.6 and 4.1.

Theorem 4.13. Let \( G \) be a digraph with \( n \) vertices and \( m \) edges. We can compute the vertex-resilient blocks of \( G \) in \( O(m+n) \) time. Also, given the vertex-resilient blocks of \( G \), we can test in \( O(1) \) time if any two vertices are vertex-resilient.

5 Computing the 2-vertex-connected blocks

It follows from Lemma 3.7 that we can test in \( O(1) \)-time if two given vertices are 2-vertex-connected by applying the \( O(1) \)-time test of \[7\] and the \( O(1) \)-time test for vertex-resilience. Alternatively, we can compute the 2-vertex-connected blocks in \( O(n) \) time given the vertex-resilient blocks \( B \) and the 2-edge-connected blocks \( S \), simply by executing \( \text{refine}(B,S) \). Then, since the 2-vertex-connected blocks have a block forest representation, we can test if two given vertices are 2-vertex-connected in \( O(1) \) time as described in Section 3.

Theorem 5.1. Let \( G \) be a digraph with \( n \) vertices and \( m \) edges. We can compute the 2-vertex-connected blocks of \( G \) in \( O(m+n) \) time. Also, given the 2-vertex-connected blocks of \( G \), we can test in \( O(1) \) time if any two vertices are 2-vertex-connected.

6 Sparse certificate for the vertex-resilient blocks and the 2-vertex-connected blocks

Here we show how to extend Algorithm FastVRB so that it also computes in linear time a sparse certificate for the vertex-resilient and the 2-vertex-connected relations. That is, we compute a subgraph \( C(G) \) of the input graph \( G \) that has \( O(n) \) edges and maintains the same vertex-resilient and 2-vertex-connected blocks as the input graph. We can achieve this by applying the same approach we used in \[7\] for computing a sparse certificate for the 2-edge-connected blocks.

As in Section 4 we can assume without loss of generality that \( G \) is strongly connected, in which case subgraph \( C(G) \) will also be strongly connected. The certificate uses the concept of independent
spanning trees \cite{9}. 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 independent if for all $v$, the paths from $s$ to $v$ in $B$ and $R$ share only the dominators of $v$. Every flow graph $G(s)$ has two such spanning trees, computable in linear time \cite{9}. Moreover, the computed spanning trees are maximally edge-disjoint, meaning that the only edges they have in common are the bridges of $G(s)$.

During the execution of Algorithm \text{FastVRB}, 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 the empty. During Step 1 of Algorithm \text{FastVRB} we compute two independent 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 independent 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 3.5, we compute two spanning trees for every connected component $S_i$ of each auxiliary graph $H^R_q \setminus q$ 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 $H^R_{S_i}(v)$. We insert in $L$ the original edges that correspond to the edges of these spanning trees.

**Lemma 6.1.** The sparse certificate $C(G)$ has the same vertex-resilient blocks and 2-vertex-connected blocks as the input digraph $G$.

**Proof.** We first argue that the execution of Algorithm \text{FastVRB} on $C(G)$ and produces the same vertex-resilient blocks as the execution of Algorithm \text{FastVRB} on $G$. The correctness of Algorithm \text{FastVRB} 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 \text{FastVRB} with input $G$ (resp. $C(G)$) as \text{FastVRB}(G) (resp. \text{FastVRB}(C(G))).

First we note that $C(G)$ is strongly connected since it contains a spanning tree of $G(s)$ and a spanning tree for the reverse of each auxiliary graph $G_r$. Moreover, the fact that $C(G)$ contains two independent spanning trees of $G$ implies that $G$ and $C(G)$ have the same dominator tree with respect to the start vertex $s$ that are computed in Step 1. Hence, the auxiliary graphs computed in Step 2 of Algorithm \text{FastVRB} have the same sets of ordinary and auxiliary vertices in both executions \text{FastVRB}(G) and \text{FastVRB}(C(G)). Hence, Step 3.1 computes the same dominator trees $D_H(r)$ and $D_H^R(r)$ in both executions, and therefore Steps 3.2 and 3.3 compute the same blocks. The same argument as in Steps 1 and 2 implies that both executions \text{FastVRB}(G) and \text{FastVRB}(C(G)) compute in Step 3.4 auxiliary graphs $H^R_q$ with the same sets of ordinary and auxiliary vertices. Finally, by construction, the strongly connected components of each auxiliary graph $H^R_q \setminus q$ are the same in both executions of \text{FastVRB}(G) and \text{FastVRB}(C(G)).

We conclude that \text{FastVRB}(G) and \text{FastVRB}(C(G)) compute the same vertex-resilient blocks as claimed. Next, observe that since the independent spanning trees computed in Steps 1 and 3.1 of the extended version of \text{FastVRB} are maximally edge-disjoint, $C(G)$ maintains the same strong bridges as $G$. Then, by Lemma \[3.7\] $C(G)$ also has the same 2-vertex-connected blocks as $G$. 

**References**

[1] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
[2] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications (Springer Monographs in Mathematics).* Springer, 1st ed. 2001. 3rd printing edition, 2002.

[3] 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.

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

[5] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms, 23*-2–20, 2013.

[6] H. N. Gabow. A poset approach to dominator computation. Unpublished manuscript 2010, revised unpublished manuscript, 2013.

[7] 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, 2015.* To appear. Preliminary version at [http://arxiv.org/abs/1407.3041](http://arxiv.org/abs/1407.3041).

[8] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms,* pages 862–871, 2004.

[9] L. Georgiadis and R. E. Tarjan. Dominator tree certification and independent spanning trees. *CoRR, abs/1210.8303,* 2012.

[10] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science, 447*(0):74–84, 2012.

[11] R. Jaberi. Computing the 2-blocks of directed graphs. *CoRR, abs/1407.6178,* 2014.

[12] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *CoRR, abs/1401.6000,* 2014.

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

[14] K. Menger. Zur allgemeinen kurventheorie. *Fund. Math.,* 10:96–115, 1927.

[15] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity.* Cambridge University Press, 2008. 1st edition.

[16] J. H. Reif and P. G. Spirakis. Strong $k$-connectivity in digraphs and random digraphs. Technical Report TR-25-81, Harvard University, 1981.

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

[18] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing, 3*(1):62–89, 1974.

[19] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM, 22*(2):215–225, 1975.
[20] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.