On Low-High Orders of Directed Graphs: An Incremental Algorithm and Applications

Loukas Georgiadis∗ Aikaterini Karanasiou∗ Giannis Konstantinos∗ Luigi Laura†

August 24, 2016

Abstract

A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex $s$. The dominator tree $D$ of $G$ is a tree rooted at $s$, such that a vertex $v$ is an ancestor of a vertex $w$ if and only if all paths from $s$ to $w$ include $v$. The dominator tree is a central tool in program optimization and code generation, and has many applications in other diverse areas including constraint programming, circuit testing, biology, and in algorithms for graph connectivity problems. A low-high order of $G$ is a preorder $\delta$ of $D$ that certifies the correctness of $D$, and has further applications in connectivity and path-determination problems. In this paper we first consider how to maintain efficiently a low-high order of a flow graph incrementally under edge insertions. We present algorithms that run in $O(mn)$ total time for a sequence of $m$ edge insertions in an initially empty flow graph with $n$ vertices. These immediately provide the first incremental certifying algorithms for maintaining the dominator tree in $O(mn)$ total time, and also imply incremental algorithms for other problems. Hence, we provide a substantial improvement over the $O(m^2)$ simple-minded algorithms, which recompute the solution from scratch after each edge insertion. We also show how to apply low-high orders to obtain a linear-time 2-approximation algorithm for the smallest 2-vertex-connected spanning subgraph problem (2VCSS). Finally, we present efficient implementations of our new algorithms for the incremental low-high and 2VCSS problems, and conduct an extensive experimental study on real-world graphs taken from a variety of application areas. The experimental results show that our algorithms perform very well in practice.

1 Introduction

A flow graph $G = (V, E, s)$ is a directed graph (digraph) with a distinguished start vertex $s \in V$. A vertex $v$ is reachable in $G$ if there is a path from $s$ to $v$; $v$ is unreachable if no such path exists. The dominator relation in $G$ is defined for the set of reachable vertices as follows. A vertex $v$ is a dominator of a vertex $w$ ($v$ dominates $w$) if every path from $s$ to $w$ contains $v$; $v$ is a proper dominator of $w$ if $v$ dominates $w$ and $v \neq w$. The dominator relation in $G$ can be represented by a tree rooted at $s$, the dominator tree $D$, such that $v$ dominates $w$ if and only if $v$ is an ancestor of $w$ in $D$. Lengauer and Tarjan [35] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with $n$ vertices and $m$ edges, where $\alpha$ is a functional inverse of Ackermann’s function [45]. Subsequently, several linear-time algorithms were discovered [3, 9, 14, 15]. The dominator tree is a central

∗Department of Computer Science & Engineering, University of Ioannina, Greece. E-mail: loukas@cs.uoi.gr, akaranas@cs.uoi.gr, giannis_konstantinos@outlook.com.
†Dipartimento di Ingegneria Informatica, Automatica e Gestionale, “Sapienza” Università di Roma, Italy. E-mail: laura@dis.uniroma1.it.
Figure 1: A flow graph $G$, its dominator tree $D$, and two strongly divergent spanning trees $B$ and $R$. The numbers correspond to a preorder numbering of $D$ that is a low-high order of $G$.

A low-high order $\delta$ of $G$ [25] is a preorder of the dominator tree $D$ such for all reachable vertices $v \neq s$, $(d(v), v) \in E$ or there are two edges $(u, v), (w, v) \in E$ such that $u$ and $w$ are reachable, $u$ is less than $v$ ($u <_\delta v$), $v$ is less than $w$ ($v <_\delta w$), and $w$ is not a descendant of $v$ in $D$. See Figure 1. Every flow graph $G$ has a low-high order, computable in linear-time [25]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [48]. By augmenting an algorithm that computes the dominator tree $D$ of a flow graph $G$ so that it also computes a low-high order of $G$, one obtains a certifying algorithm to compute $D$. (A certifying algorithm [37] outputs both the solution and a correctness certificate, with the property that it is straightforward to use the certificate to verify that the computed solution is correct.) Low-high orders also have applications in path-determination problems [47] and in fault-tolerant network design [6, 7, 26].

A notion closely related to low-high orders is that of divergent spanning trees [25]. Let $V_r$ be the set of reachable vertices, and let $G[V_r]$ be the flow graph with start vertex $s$ that is induced by $V_r$. Two spanning trees $B$ and $R$ of $G[V_r]$, rooted at $s$, are divergent if for all $v$, the paths from $s$
to \( v \) in \( B \) and \( R \) share only the dominators of \( v \); \( B \) and \( R \) are strongly divergent if for every pair of vertices \( v \) and \( w \), either the path in \( B \) from \( s \) to \( v \) and the path in \( R \) from \( s \) to \( w \) share only the common dominators of \( v \) and \( w \), or the path in \( R \) from \( s \) to \( v \) and the path in \( B \) from \( s \) to \( w \) share only the common dominators of \( v \) and \( w \). In order to simplify our notation, we will refer to \( B \) and \( R \), with some abuse of terminology, as strongly divergent spanning trees of \( G \). Every flow graph has a pair of strongly divergent spanning trees. Given a low-high order of \( G \), it is straightforward to compute two strongly divergent spanning trees of \( G \) in \( O(m) \) time [25]. Divergent spanning trees can be used in data structures that compute pairs of vertex-disjoint \( s\)-\( t \) paths in 2-vertex connected digraphs (for any two query vertices \( s \) and \( t \)) [19], in fast algorithms for approximating the smallest 2-vertex-connected spanning subgraph of a digraph [20], and in constructing sparse subgraphs of a given digraph that maintain certain connectivity requirements [21, 32, 33].

In this paper we consider how to update a low-high order of a flow graph through a sequence of edge insertions. See Figure 2. The difficulty in updating the dominator tree and a low-high order is due to the following facts. An affected vertex can be arbitrarily far from the inserted edge, and a single edge insertion may cause \( O(n) \) parent changes in \( D \). Furthermore, since a low-high order is a preorder of \( D \), a single edge insertion may cause \( O(n) \) changes in this order, even if there is only one vertex that is assigned a new parent in \( D \) after the insertion. More generally,
we note that the hardness of dynamic algorithms on digraphs has been recently supported also by conditional lower bounds [1]. Our first contribution is to show that we can maintain a low-high order of a flow graph \( G \) with \( n \) vertices through a sequence of \( m \) edge insertions in \( O(mn) \) total time. Hence, we obtain a substantial improvement over the naive solution of recomputing a low-high order from scratch after each edge insertion, which takes \( O(m^2) \) total time. Our result also implies the first incremental certifying algorithms [37] for computing dominators in \( O(mn) \) total time, which answers an open question in [25]. We present two algorithms that achieve this bound, a simple and a more sophisticated. Both algorithms combine the incremental dominators algorithm of [23] with the linear-time computation of two divergent spanning trees from [25]. Our more sophisticated algorithm also applies a slightly modified version of a static low-high algorithm from [25] on an auxiliary graph. Although both algorithms have the same worst-case running time, our experimental results show that the sophisticated algorithm is by far superior in practical scenarios.

We note that the incremental dominators problem arises in various applications, such as incremental data flow analysis and compilation [11, 18, 41, 43], distributed authorization [38], and in incremental algorithms for maintaining 2-connectivity relations in directed graphs [24]. In Section 4 we show how our result on incremental low-high order maintenance implies the following incremental algorithms that also run in \( O(mn) \) total time for a sequence of \( m \) edge insertions.

- First, we give an algorithm that maintains, after each edge insertion, two strongly divergent spanning trees of \( G \), and answers the following queries in constant time: (i) For any two query vertices \( v \) and \( w \), find a path \( \pi_{sv} \) from \( s \) to \( v \) and a path \( \pi_{sw} \) from \( s \) to \( w \), such that \( \pi_{sv} \) and \( \pi_{sw} \) share only the common dominators of \( v \) and \( w \). We can output these paths in \( O(|\pi_{sv}| + |\pi_{sw}|) \) time. (ii) For any two query vertices \( v \) and \( w \), find a path \( \pi_{sv} \) from \( s \) to \( v \) that avoids \( w \), if such a path exists. We can output this path in \( O(|\pi_{sv}|) \) time.

- Then we provide an algorithm for an incremental version of the fault-tolerant reachability problem [6, 7]. We maintain a flow graph \( G = (V, E, s) \) with \( n \) vertices through a sequence of \( m \) edge insertions, so that we can answer the following query in \( O(n) \) time. Given a spanning forest \( F = (V, E_F) \) of \( G \) rooted at \( s \), find a set of edges \( E' \subseteq E \setminus E_F \) of minimum cardinality, such that the subgraph \( G' = (V, E_F \cup E', s) \) of \( G \) has the same dominators as \( G \).

- Finally, given a digraph \( G \), we consider how to maintain incrementally a spanning subgraph of \( G \) with \( O(n) \) edges that preserves the 2-edge-connectivity relations in \( G \).

We also revisit the problem of computing a smallest 2-vertex-connected spanning subgraph (2VCSS) of a directed graph [10, 20]. We present a linear-time algorithm that computes a 2-approximation of the smallest 2-vertex-connected spanning subgraph (2VCSS) of a 2-vertex-connected digraph. This improves significantly the best previous approximation ratio achievable in linear time for this problem, which was 3 [20]. Our new algorithm is also shown to achieve better performance in practice.

2 Preliminaries

Let \( G = (V, E, s) \) be a flow graph with start vertex \( s \), and let \( D \) be the dominator tree of \( G \). A spanning tree \( T \) of \( G \) is a tree with root \( s \) that contains a path from \( s \) to \( v \) for all reachable vertices \( v \). We refer to a spanning subgraph \( F \) of \( T \) as a spanning forest of \( G \). Given a rooted tree \( T \), we denote by \( T[v] \) the subtree of \( T \) rooted at \( v \) (we also view \( T(v) \) as the set of descendants of \( v \)). Let \( T \) be a tree rooted at \( s \) with vertex set \( V_T \subseteq V \), and let \( t(v) \) denote the parent of a vertex \( v \in V_T \) in \( T \). If \( v \) is an ancestor of \( w \), \( T[v, w] \) is the path in \( T \) from \( v \) to \( w \). In particular, \( D[s, v] \) consists of
the vertices that dominate \( v \). If \( v \) is a proper ancestor of \( w \), \( T(v, w) \) is the path to \( w \) from the child of \( v \) that is an ancestor of \( w \). Tree \( T \) is flat if its root is the parent of every other vertex. Suppose now that the vertex set \( V_T \) of \( T \) consists of the vertices reachable from \( s \). Tree \( T \) has the parent property if for all \((v, w) \in E\) with \( v \) and \( w \) reachable, \( v \) is a descendant of \( t(w) \) in \( T \). If \( T \) has the parent property and has a low-high order, then \( T = D \) [25]. For any vertex \( v \in V \), we denote by \( C(v) \) the set of children of \( v \) in \( D \). A preorder of \( T \) is a total order of the vertices of \( T \) such that, for every vertex \( v \), the descendants of \( v \) are ordered consecutively, with \( v \) first.

Let \( \zeta \) be a preorder of \( D \). Consider a vertex \( v \neq s \). We say that \( \zeta \) is a low-high order for \( v \) in \( G \), if \((d(v), v) \in E\) or there are two edges \((u, v), (w, v) \in E\) such that \( u <_\zeta v \) and \( v <_\zeta w \), and \( w \) is not a descendant of \( v \) in \( D \). Given a graph \( G = (V, E) \) and a set of edges \( S \subseteq V \times V \), we denote by \( G \cup S \) the graph obtained by inserting into \( G \) the edges of \( S \).

3 Incremental low-high order

In this section we describe two algorithms to maintain a low-high order of a digraph through a sequence of edge insertions. We first review some useful facts for updating a dominator tree after an edge insertion [4, 23, 41]. Let \((x, y)\) be the edge to be inserted. We consider the effect of this insertion when both \( x \) and \( y \) are reachable. Let \( G' \) be the flow graph that results from \( G \) after inserting \((x, y)\). Similarly, if \( D \) is the dominator tree of \( G \) before the insertion, we let \( D' \) be the the dominator tree of \( G' \). Also, for any function \( f \) on \( V \), we let \( f' \) be the function after the update. We say that vertex \( v \) is affected by the update if \( d(v) \) (its parent in \( D \)) changes, i.e., \( d'(v) \neq d(v) \). We let \( A \) denote the set of affected vertices. Note that we can have \( D'[s, v] \neq D[s, v] \) even if \( v \) is not affected. We let \( nca(x, y) \) denote the nearest common ancestor of \( x \) and \( y \) in the dominator tree \( D \).

We also denote by \( \text{depth}(v) \) the depth of a reachable vertex \( v \) in \( D \). There are affected vertices after the insertion of \((x, y)\) if and only if \( nca(x, y) \) is not a descendant of \( d(y) \) [42]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [4].

**Lemma 3.1.** ([23]) Suppose \( x \) and \( y \) are reachable vertices in \( G \). A vertex \( v \) is affected after the insertion of edge \((x, y)\) if and only if \( \text{depth}(nca(x, y)) < \text{depth}(d(v)) \) and there is a path \( \pi \) in \( G \) from \( y \) to \( v \) such that \( \text{depth}(d(v)) < \text{depth}(w) \) for all \( w \in \pi \). If \( v \) is affected, then it becomes a child of \( nca(x, y) \) in \( D' \), i.e., \( d'(v) = nca(x, y) \).

The algorithm (DBS) in [23] applies Lemma 3.1 to identify affected vertices by starting a search from \( y \) (if \( y \) is not affected, then no other vertex is). To do this search for affected vertices, it suffices to maintain the outgoing and incoming edges of each vertex. These sets are organized as singly linked lists, so that a new edge can be inserted in \( O(1) \) time. The dominator tree \( D \) is represented by the parent function \( d \). We also maintain the depth in \( D \) of each reachable vertex. We say that a vertex \( v \) is scanned, if the edges leaving \( v \) are examined during the search for affected vertices, and that it is visited if there is a scanned vertex \( u \) such that \((u, v)\) is an edge in \( G \). By Lemma 3.1, a visited vertex \( v \) is scanned if \( \text{depth}(nca(x, y)) < \text{depth}(d(v)) \).

**Lemma 3.2.** ([23]) Let \( v \) be a scanned vertex. Then \( v \) is a descendant of an affected vertex in \( D \).

3.1 Simple Algorithm

In this algorithm we maintain, after each insertion, a subgraph \( H = (V, E_H) \) of \( G \) with \( O(n) \) edges that has the same dominator tree as \( G \). Then, we can compute a low-high order \( \delta \) of \( H \) in \( O(|E_H|) = O(n) \) time. Note that \( \delta \) is also a valid low-high order of \( G \). Subgraph \( H \) is formed by the edges of two divergent spanning trees \( B \) and \( R \) of \( G \). After the insertion of an edge \((x, y)\),
where both $x$ and $y$ are reachable, we form a graph $H'$ by inserting into $H$ a set of edges $\text{Last}(A)$ found during the search for affected vertices. Specifically, $\text{Last}(A)$ contains edge $(x, y)$ and, for each affected vertex $v \neq y$, the last edge on a path $\pi_{yv}$ that satisfies Lemma 3.1. Then, we set $H' = H \cup \text{Last}(A)$. Finally, we compute a low-high order and two divergent spanning trees of $H'$, which are also valid for $G'$. Algorithm \text{SimpleInsertEdge} describes this process.

Algorithm 1: \text{SimpleInsertEdge}(G, D, \delta, B, R, e)

| Input: Flow graph $G = (V, E, s)$, its dominator tree $D$, a low-high order $\delta$ of $G$, two divergent spanning trees $B$ and $R$ of $G$, and a new edge $e = (x, y)$. |
|----------------------------------|
| Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree $D'$, a low-high order $\delta'$ of $G'$, and two divergent spanning trees $B'$ and $R'$ of $G'$. |
|----------------------------------|

1. Insert $e$ into $G$ to obtain $G'$.
2. If $x$ is unreachable in $G$ then return $(G', D, \delta, B, R)$ else if $y$ is unreachable in $G$ then
3. 
   $(D', \delta', B', R') \leftarrow \text{ComputeLowHigh}(G')$
4. return $(G', D', \delta', B', R')$
5. end
6. Let $H = B \cup R$.
7. Compute the updated dominator tree $D'$ of $G'$ and return a list $A$ of the affected vertices, and a list $\text{Last}(A)$ of the last edge entering each $v \in A$ in a path of Lemma 3.2.
8. Compute the subgraph $H' = H \cup \text{Last}(A)$ of $G'$.
9. Compute $(D', \delta', B', R') \leftarrow \text{ComputeLowHigh}(H')$
10. return $(G', D', \delta', B', R')$

Note that when only $x$ is reachable before the insertion, we re-initialize our algorithm by running the linear-time algorithm of [25, Section 6], which returns both a low-high order and two divergent spanning trees.

Lemma 3.3. Algorithm \text{SimpleInsertEdge} is correct.

Proof. It suffices to show that subgraph $H'$ of $G'$, computed in line 9, has the same dominator tree with $G'$. Note that graph $H$, formed by two divergent spanning trees $B$ and $R$ of $G$ in line 7, has the same dominator tree $D$ as $G$. Hence, since $\text{Last}(A)$ contains $(x, y)$, the immediate dominator of $y$ is the same in $H'$ and in $G'$. Let $A$ be the set of affected vertices in $G$ after the insertion of edge $(x, y)$. Since $H'$ is a subgraph of $G'$, any vertex in $V \setminus A$ has the same immediate dominator in $H'$ and in $G'$. It remains to argue that for each vertex $v \in A \setminus y$, there is a path $\pi_{yv}$ in $H'$ that satisfies Lemma 3.1. Let $\pi_{yv}$ be the path from $y$ in $v$ in $G$ that was found by the search for affected vertices performed by the algorithm of [23]. We give a corresponding path $\hat{\pi}_{yv}$ in $H'$. Recall that every vertex on $\pi_{yv}$ is scanned and that every scanned vertex is a descendant in $D$ of an affected vertex. We argue that for every two successive affected vertices $u$ and $w$ on $\pi_{yv}$ there is a path $\pi_{uw}$ from $u$ to $w$ in $H'$ that consists of vertices of depth at least $\text{depth}(w)$. Note that, by properties of the depth-based search, $\text{depth}(w) \leq \text{depth}(u)$. Indeed, let $(p, w)$ be the edge entering $w$ from $\pi_{yv}$. Then $(p, w) \in \text{Last}(A)$ and $p$ is a descendant of $u$ in $D$. Also, since $u$ dominates $p$ in $G$, $u$ is an ancestor of $p$ in both spanning trees $B$ and $R$. We let $\hat{\pi}_{uw} = B[u, p] \cdot (p, w)$. All vertices on $B[u, p]$ are dominated by $u$, since otherwise there would be a path from $s$ to $p$ avoiding $u$. So, $\hat{\pi}_{uw}$ is path from $u$ to $w$ in $H'$ that consists of vertices with depth at least $\text{depth}(w)$.

Lemma 3.4. Algorithm \text{SimpleInsertEdge} maintains a low-high order of a flow graph $G$ with $n$ vertices through a sequence of edge insertions in $O(nm)$ total time, where $m$ is the total number of edges in $G$ after all insertions.
Proof. Consider the insertion of an edge \((x, y)\). If \(y\) was unreachable in \(G\) then we compute \(D\), two divergent spanning trees \(B\) and \(R\), and a low-high order in \(O(m)\) time [25]. Throughout the whole sequence of \(m\) insertions, such an event can happen \(O(n)\) times, so all insertions to unreachable vertices are handled in \(O(mn)\) total time.

Now we consider the cost of executing SimpleInsertEdge, when both \(x\) and \(y\) are reachable in \(G\). Let \(\nu\) be the number of scanned vertices, and let \(\mu\) be the number of their adjacent edges. We can update the dominator tree and locate the affected vertices (line 8) in \(O(\nu + \mu + n)\) time [23]. At the same time we can compute the edge set \(\text{Last}(A)\). Computing \(H'\) in line 9 takes \(O(n)\) time since \(B \cup R \cup \text{Last}(A)\) contains at most \(3(n - 1)\) edges. Also, computing the dominator tree, two divergent spanning trees,, and a low-high order of \(H'\) in \(O(n)\) time [25]. So SimpleInsertEdge runs in \(O(\nu + \mu + n)\) time. . The \(O(n)\) term gives a total cost of \(O(mn)\) for the whole sequence of \(m\) insertions. We distribute the remaining \(O(\nu + \mu)\) cost to the scanned vertices and edges, that is \(O(1)\) per scanned vertex or edge. Since the depth in \(D\) of every scanned vertex decreases by at least one, a vertex and an edge can be scanned at most \(O(n)\) times. Hence, each vertex and edge can contribute at most \(O(n)\) total cost through the whole sequence of \(m\) insertions. The \(O(mn)\) bound follows.

### 3.2 Efficient Algorithm

Here we develop a more practical algorithm that maintains a low-high order \(\delta\) of a flow graph \(G = (V, E, s)\) through a sequence of edge insertions. Our algorithm uses the incremental dominators algorithm of [23] to update the dominator tree \(D\) of \(G\) after each edge insertion. We describe a process to update \(\delta\) based on new results on the relation among vertices in \(D\) that are affected by the insertion. . These results enable us to identify a subset of vertices for which we can compute a “local” low-high order, that can be extended to a valid low-high order of \(G\) after the update. We show that such a “local” low-high order can be computed by a slightly modified version of an algorithm from [25]. We apply this algorithm on a sufficiently small flow graph that is defined by the affected vertices, and is constructed using the concept of derived edges [46].

#### 3.2.1 Derived edges and derived flow graphs

Derived graphs, first defined in [46], reduce the problem of finding a low-high order to the case of a flat dominator tree [25]. By the parent property of \(D\), if \((v, w)\) is an edge of \(G\), the parent \(d(w)\) of \(w\) is an ancestor of \(v\) in \(D\). Let \((v, w)\) be an edge of \(G\), with \(w\) not an ancestor of \(v\) in \(D\). Then, the derived edge of \((v, w)\) is the edge \((\overline{v}, w)\), where \(\overline{v} = v\) if \(v = d(w)\), \(\overline{v}\) is the sibling of \(w\) that is an ancestor of \(v\) if \(v \neq d(w)\). If \(w\) is an ancestor of \(v\) in \(D\), then the derived edge of \((v, w)\) is null. Note that a derived edge \((\overline{v}, w)\) may not be an original edge of \(G\). For any vertex \(w \in V\) such that \(C(w) \neq \emptyset\), we define the derived flow graph of \(w\), denoted by \(G_w = (V_w, E_w, w)\), as the flow graph with start vertex \(w\), vertex set \(V_w = C(w) \cup \set{w}\), and edge set \(E_w = \set{(\overline{v}, v) \mid v \in V_w\ and \ (\overline{v}, v) \ is \ the \ non-null \ derived \ edge \ of \ some \ edge \ in \ E}\). By definition, \(G_w\) has flat dominator tree, that is, \(w\) is the only proper dominator of any vertex \(v \in V_w\ \setminus w\). We can compute a low-high order \(\delta\) of \(G\) by computing a low-high order \(\delta_{w}\) in each derived flow graph \(G_w\). Given these low-high orders \(\delta_{w}\), we can compute a low-high order of \(G\) in \(O(n)\) time by a depth-first traversal of \(D\). During this traversal, we visit the children of each vertex \(w\) in their order in \(\delta_{w}\), and number the vertices from 1 to \(n\) as they are visited. The resulting preorder of \(D\) is low-high on \(G\). Our incremental algorithm identifies, after each edge insertion, a specific derived flow graph \(G_w\) for which a low-high order \(\delta_{w}\) needs to be updated. Then, it uses \(\delta_{w}\) to update the low-high order of the whole flow graph \(G\). Still, computing a low-high order of \(G_w\) can
be too expensive to give us the desired running time. Fortunately, we can overcome this obstacle by exploiting a relation among the vertices that are affected by the insertion, as specified below. This allows us to compute $\delta_w$ in a contracted version of $G_w$.

### 3.2.2 Affected vertices

Let $(x, y)$ be the inserted vertex, where both $x$ and $y$ are reachable. Consider the execution of algorithm DBS [23] that updates the dominator tree by applying Lemma 3.1. Suppose vertex $v$ is scanned, and let $q$ be the nearest affected ancestor of $v$ in $D$. Then, by Lemma 3.1, vertex $q$ is a child of $nca(x, y)$ in $D'$, i.e., $d'(q) = nca(x, y)$, and $v$ remains a descendant of $q$ in $D'$.

**Lemma 3.5.** Let $u$ and $v$ be vertices such that $u \in D(v)$. Then, any simple path from $v$ to $u$ in $G$ contains only vertices in $D(v)$.

**Proof.** Since $u \in D(v)$, $v$ dominates $u$, so all paths from $s$ to $u$ contain $v$. Let $\pi_{vu}$ be a simple path from $v$ to $u$. Suppose, for contradiction, that $\pi_{vu}$ contains a vertex $w \notin D(v)$. Let $\pi_{wu}$ be the part of $\pi_{vu}$ from $w$ to $u$. Since $w \notin D(v)$, there is a path $\pi_{sw}$ from $s$ to $w$ that avoids $v$. But then $\pi_{sw} \cdot \pi_{wu}$ is a path from $s$ to $u$ that avoids $v$, a contradiction. \hfill \Box

**Lemma 3.6.** Let $v$ be a vertex that is affected by the insertion of $(x, y)$, and let $w$ be a sibling of $v$ in $D$. If there is an edge $(u, w)$ with $u$ a descendant of $v$ in $D$ then $w$ is also affected.

**Proof.** Since $v$ is affected, there is a path $\pi_{yw}$ from $y$ to $v$ in $G$ that satisfies Lemma 3.1. By Lemma 3.5 and the fact that $u$ is a descendant of $v$ in $D$, there is a simple path $\pi_{vu}$ from $v$ to $u$ in $G$ that contains only vertices in $D(v)$. Thus, $\pi_{yw} \cdot \pi_{vu} \cdot (u, w)$ is a path from $y$ to $w$ that also satisfies Lemma 3.1. Hence, $w$ is affected. \hfill \Box

**Lemma 3.7.** Let $v$ be an ancestor of $w$ in $D$, and let $u$ be a vertex that is not a descendant of $v$ in $D$. Then any path from $u$ to $w$ contains $v$.

**Proof.** Let $\pi_{uw}$ be a path from $u$ to $w$. Since $u$ is not a descendant of $v$, there is a path $\pi_{sw}$ from $s$ to $u$ that avoids $v$. Hence, if $\pi_{uw}$ does not contain $v$, then $\pi_{su} \cdot \pi_{uw}$ is path from $s$ to $w$ that avoids $v$, a contradiction. \hfill \Box

Our next lemma provides a key result about the relation of the affected vertices in $D$.

**Lemma 3.8.** All vertices that are affected by the insertion of $(x, y)$ are descendants of a common child $c$ of $nca(x, y)$.

**Proof.** Let $z = nca(x, y)$, and let $c$ be the child of $z$ that is an ancestor of $y$ in $D$. We claim that all affected vertices are descendants of $c$ in $D$. Suppose, for contradiction, that there is an affected vertex $v$ that is not a descendant of $c$ in $D$. By Lemma 3.1, $v$ must be a descendant $z$ in $D$. Also, since the children of $z$ are not affected, $v$ is not a child of $z$. Hence, $v$ is a proper descendant of another child $q$ of $z$ in $D$ ($q \neq c$). Let $\pi_{yz}$ be a path from $y$ to $v$ in $G$ that satisfies Lemma 3.1. Since $y$ is not a descendant of $q$, by Lemma 3.7 path $\pi_{yz}$ must contain $q$. But then $\pi_{yz}$ contains a vertex of depth $\text{depth}(d(v))$ or less, which contradicts Lemma 3.1. \hfill \Box

We shall apply Lemma 3.8 to construct a flow graph $G_A$ for the affected vertices. Then, we shall use $G_A$ to compute a “local” low-high order that we extend to a valid low-high order of $G'$. 

8
3.2.3 Low-high order augmentation

Let \( \delta \) be a low-high order of \( G \), and let \( \delta' \) be a preorder of the dominator tree \( D' \) of \( G' \). We say that \( \delta' \) agrees with \( \delta \) if the following condition holds for any pair of siblings \( u, v \) in \( D \) that are not affected by the insertion of \( (x, y) \): \( u <_{\delta'} v \) if and only if \( u <_\delta v \). Our goal is to show that there is a low-high order \( \delta' \) of \( G' \) that agrees with \( \delta \).

**Lemma 3.9.** Let \( \delta \) be a low-high order of \( G \) before the insertion of \( (x, y) \). There is a preorder \( \delta' \) of \( D' \) that agrees with \( \delta \).

*Proof.* By Lemma 3.1, all affected vertices become children of \( z \) in \( D' \). Hence, \( C'(z) \supseteq C(z) \), and for any \( v \neq z \), \( C'(v) \subseteq C(v) \). Then, for each vertex \( v \), we can order the children of \( v \) in \( C'(v) \) that are not affected according to \( \delta \). Finally, we insert the affected vertices in any order in the list of children of \( z \). Let \( \delta' \) be the preorder of \( D' \) that is constructed by a depth-first traversal of \( D' \) that visits the children of each vertex \( w \) in the order specified above. Then, \( \delta' \) agrees with \( \delta \). \( \square \)

**Lemma 3.10.** Let \( \delta' \) be a preorder of \( D' \) that agrees with \( \delta \). Let \( v \) be a vertex that is not a child of \( nca(x, y) \) and is not affected by the insertion of \( (x, y) \). Then \( \delta' \) is a low-high order for \( v \) in \( G' \).

*Proof.* Since \( v \) is not affected, \( d(v) \) is still the parent of \( v \) in \( D \) after the insertion. So, if \( (d(v), v) \in E \), then \( \delta' \) is a low-high order for \( v \) in \( G' \). Now suppose that \( (d(v), v) \notin E \). Then there are two edges \( (u, v) \) and \( (w, v) \) in \( E \) such that \( u <_\delta v <_\delta w \), where \( w \) is not a descendant of \( v \) in \( D \). Let \( (\overline{u}, v) \) and \( (\overline{w}, v) \) be the derived edges of \( (u, v) \) and \( (w, v) \), respectively, in \( D \). Then \( \overline{u} \) and \( \overline{w} \) are siblings of \( v \) in \( D \). Siblings \( \overline{u} \) and \( \overline{w} \) exist and are distinct by the fact that \( (d(v), v) \notin E \) and by the parent property of \( D \). Hence, \( \overline{u} <_\delta v <_\delta \overline{w} \). We argue that after the insertion of \( (x, y) \), \( \overline{u} \) (resp., \( \overline{w} \)) remains a sibling of \( v \), and an ancestor of \( u \) (resp., \( w \)). If this is not the case, then there is an affected vertex \( q \) on \( D[\overline{u}, u] \). But then, Lemma 3.6 implies that \( v \) is also affected, a contradiction. So, both \( \overline{u} \) and \( \overline{w} \) remain siblings of \( v \) in \( D' \), and \( (\overline{u}, v) \) and \( (\overline{w}, v) \) remain the derived edges of \( (u, v) \) and \( (w, v) \), respectively, in \( D' \). Then, since \( \delta' \) agrees with \( \delta \), \( \delta' \) is a low-high order for \( v \) in \( G' \). \( \square \)

We shall use Lemmata 3.1 and 3.10 to show that in order to compute a low-high order of \( G' \), it suffices to compute a low-high order for the derived flow graph \( G'_z \), where \( z = nca(x, y) \). Still, the computation of a low-high order of \( G'_z \) is too expensive to give us the desired running time. Fortunately, as we show next, we can limit these computations for a contracted version of \( G'_z \), defined by the affected vertices.

Let \( \delta \) be a low-high order of \( G \) before the insertion of \( (x, y) \). Also, let \( z = nca(x, y) \), and let \( \delta_z \) be a corresponding low-high order of the derived flow graph \( G_z \). That is, \( \delta_z \) is the restriction of \( \delta \) to \( z \) and its children in \( D \). Consider the child \( c \) of \( z \) that, by Lemma 3.8, is an ancestor of all the affected vertices. Let \( \alpha \) and \( \beta \), respectively, be the predecessor and successor of \( c \) in \( \delta_z \). Note that \( \alpha \) or \( \beta \) may be null. An augmentation of \( \delta_z \) is an order \( \delta'_z \) of \( C'(z) \cup \{z\} \) that results from \( \delta_z \) by inserting the affected vertices arbitrarily around \( c \), that is, each affected vertex is placed in an arbitrary position between \( \alpha \) and \( c \) or between \( c \) and \( \beta \).

**Lemma 3.11.** Let \( z = nca(x, y) \), and let \( \delta_z \) be a low-high order of the derived flow graph \( G_z \) before the insertion of \( (x, y) \). Also, let \( \delta'_z \) be an augmentation of \( \delta_z \), and let \( \delta' \) be a preorder of \( D' \) that extends \( \delta'_z \). Then, for each child \( v \) of \( z \) in \( D \), \( \delta' \) is a low-high order for \( v \) in \( G' \).

*Proof.* Since \( v \) is a child of \( z \) in \( D \) it is not affected. Hence, \( d'(v) = d(v) = z \). Let \( G'_z \) be the derived flow graph of \( z \) after the insertion of \( (x, y) \). It suffices to show that \( \delta'_z \) is a low-high order for \( v \) in \( G'_z \).
If \((z,v) \in E\), then \((z,v)\) is an edge in \(G'_z\). So, in this case, \(\delta'_z\) is a low-high order for \(v\) in \(G'_z\).

Now suppose that \((z,v) \notin E\). Let \(\delta\) be a preorder of \(D\) that extends \(\delta_z\). Then, there are two edges \((u,v)\) and \((w,v)\) in \(G\) such that \(u <_\delta v <_\delta w\), where \(w\) is not a descendant of \(v\) in \(D\). The fact that \((z,v)\) is not an edge implies that \(u \neq z\) and \(w \neq z\). Let \(\overline{w'}\) (resp., \(\overline{w'}\)) be the nearest ancestor of \(u\) (resp., \(w\)) in \(D'\) that is a child of \(z\). We argue that \(\overline{w'}\) exists and satisfies \(\overline{w'} < _{\delta'_z} v\). Let \(\overline{u}\) be the nearest ancestor of \(u\) in \(D\). If no vertex on \(D(z,u)\) is affected, then \(\overline{u} = \delta_z\). Also, since \(\delta_z < _{\delta_z} v\) and by the fact that \(\delta'_z\) is an augmentation of \(\delta_z\), we have \(\overline{w'} < _{\delta'_z} v\). Suppose now that there is an affected vertex \(q\) on \(D(z,u)\). By Lemma 3.1, \(q\) becomes a child of \(z\) in \(D'\), hence \(\overline{w'} = q\). Also, by Lemma 3.8, \(q\) is a proper descendant of \(c\), so \(\overline{u} = c\). Then \(c < _{\delta_z} v\), and by the construction of \(\delta'_z\), we have \(\overline{w'} < _{\delta'_z} v\).

An analogous argument shows that \(\overline{w'}\) exists and satisfies \(v < _{\delta'_z} \overline{w'}\). Thus, \(\delta'_z\) is a low-high order for \(v\) in \(G'_z\).

### 3.2.4 Algorithm

Now we are ready to describe our incremental algorithm for maintaining a low-high order \(\delta\) of \(G\). For each vertex \(v\) that is not a leaf in \(D\), we maintain a list of its children \(C(v)\) in \(D\), ordered by \(\delta\). Also, for each vertex \(v \neq s\), we keep two variables \(low(v)\) and \(high(v)\). Variable \(low(v)\) stores an edge \((u,v)\) such that \(u \neq d(v)\) and \(u < _\delta v\); \(low(v) = null\) if no such edge exists. Similarly, \(high(v)\) stores an edge \((w,v)\) such that and \(v < _\delta w\) and \(w\) is not a descendant of \(v\) in \(D\); \(high(v) = null\) if no such edge exists. These variables are useful in the applications that we mention in Section 4.

Finally, we mark each vertex \(v\) such that \((d(v), v) \in E\). For simplicity, we assume that the vertices of \(G\) are numbered from 1 to \(n\), so we can store the above information in corresponding arrays \(low\), \(high\), and \(mark\). Note that for a reachable vertex \(v\), we can have \(low(v) = null\) or \(high(v) = null\) (or both) only if \(mark(v) = true\). Before any edge insertion, all vertices are unmarked, and all entries in arrays \(low\) and \(high\) are null. We initialize the algorithm and the associated data structures by executing a linear-time algorithm to compute the dominator tree \(D\) of \(G\) [3, 9] and a linear-time algorithm to compute a low-high order \(\delta\) of \(G\) [25]. So, the initialization takes \(O(m + n)\) time for a digraph with \(n\) vertices and \(m\) edges. Next, we describe the main routine to handle an edge insertion. We let \((x,y)\) be the inserted edge. Also, if \(x\) and \(y\) are reachable before the insertion, we let \(z = nca(x,y)\).

**Algorithm 2: Initialize(G)**

1. Compute the dominator tree \(D\) and a low-high order \(\delta\) of \(G\).
2. foreach reachable vertex \(v \in V \setminus s\) do
   3. if \((d(v), v) \in E\) then set \(mark(v) \leftarrow true\) find edges \((u,v)\) and \((w,v)\) such that \(u < _\delta v < _\delta w\) and \(w \notin D(v)\)
   4. set \(low(v) \leftarrow u\) and \(high(v) \leftarrow w\)
5. end
6. return \((D, \delta, mark, low, high)\)

Our main task now is to order the affected vertices according to a low-high order of \(D'\). To do this, we use an auxiliary flow graph \(G_A = (V_A, E_A, z)\), with start vertex \(z\), which we refer to as the derived affected flow graph. Flow graph \(G_A\) is essentially a contracted version of the derived flow graph \(G'_z\) (i.e., the derived graph of \(z\) after the insertion) as we explain later. The vertices of the derived affected flow graph \(G_A\) are \(z\), the affected vertices of \(G\), their common ancestor \(c\) in \(D\) that is a child of \(z\) (from Lemma 3.8), and two auxiliary vertices \(\alpha^*\) and \(\beta^*\). Vertex \(\alpha^*\) (resp.,
follows immediately from this claim. The claim is obvious for $w$.

**Proof.**

We claim that for any two distinct vertices $w$ and $\beta$ in $\delta$ of $G$, arrays $\alpha$, $\beta$, and $\gamma$, and a new edge $e = (x, y)$.

**Output:** Flow graph $G' = (V, E \cup \{x, y\}, s)$, its dominator tree $D'$, a low-high order $\delta'$ of $G'$, and arrays $\alpha'$, $\beta'$, and $\gamma'$.

1. Insert $e$ into $G$ to obtain $G'$.
2. If $x$ is unreachable in $G$ then return $(G', D, \delta, \alpha, \beta, \gamma)$ else if $y$ is unreachable in $G$ then
   3. $(D', \delta', \alpha', \beta', \gamma') \leftarrow \text{Initialize}(G')$
   4. return $(G', D', \delta', \alpha', \beta', \gamma')$
5. end
6. Compute the nearest common ancestor $z$ of $x$ and $y$ in $D$.
7. Compute the updated dominator tree $D'$ of $G'$ and return a list $A$ of the affected vertices.
8. foreach vertex $v \in A \cup \{c\}$ do
   9. if $z = x$ then $\alpha'(y) \leftarrow \text{false}$
   10. Execute $\text{DerivedLowHigh}(z, A, \alpha'(x))$.
   11. Make a dfs traversal of the subtrees of $D'$ rooted at each vertex $v \in A \cup \{c\}$ to compute $\delta'$.
   12. foreach vertex $v \in A \cup \{c\}$ do
      13. find edges $(u, v)$ and $(w, v)$ such that $u <_{\delta'} v <_{\delta'} w$ and $w \notin D'(v)$
      14. set $\alpha'(v) \leftarrow u$ and $\beta'(v) \leftarrow w$
   end
15. return $(G', D', \delta', \alpha', \beta', \gamma')$

$\beta^\ast$ represents vertices in $C(z)$ with lower (resp., higher) order in $\delta$ than $c$. We include in $G_A$ the edges $(z, \alpha^\ast)$ and $(z, \beta^\ast)$. If $c$ is marked then we include the edge $(z, c)$ into $G_A$, otherwise we add the edges $(\alpha^\ast, c)$ and $(\beta^\ast, c)$ into $G_A$. Also, for each edge $(u, v)$ such that $u$ is a descendant of an affected vertex $v$, we add in $G_A$ the edge $(v, c)$. Now we specify the edges that enter an affected vertex $w$ in $G_A$. We consider each edge $(u, w) \in E$ entering $w$ in $G$. We have the following cases:

(a) If $u$ is a descendant of an affected vertex $v$, we add in $G_A$ the edge $(v, w)$.

(b) If $u$ is a descendant of $c$ but not a descendant of an affected vertex, then we add in $G_A$ the edge $(c, w)$.

(c) If $u \neq z$ is not a descendant of $c$, then we add the edge $(\alpha^\ast, w)$ if $u <_{\delta} c$, or the edge $(\beta^\ast, w)$ if $c <_{\delta} u$.

(d) Finally, if $u = z$, then we add the edge $(z, w)$. (In cases (c) and (d), $u = x$ and $w = y$.)

See Figure 3. Recall that $\alpha$ (resp., $\beta$) is the siblings of $c$ in $D$ immediately before (resp., after) $c$ in $\delta$, if it exists. Then, we can obtain $G_A$ from $G_A'$ by contracting all vertices $v$ with $v <_{\delta} c$ into $\alpha = \alpha^\ast$, and all vertices $v$ with $c <_{\delta} v$ into $\beta = \beta^\ast$.

**Lemma 3.12.** The derived affected flow graph $G_A = (V_A, E_A, z)$ has flat dominator tree.

**Proof.** We claim that for any two distinct vertices $v, w \in V_A \backslash z$, $v$ does not dominate $w$. The lemma follows immediately from this claim. The claim is obvious for $w \in \{\alpha^\ast, \beta^\ast\}$, since $G_A$ contains the edges $(z, \alpha^\ast)$ and $(z, \beta^\ast)$. The same holds for $w = c$, since $G_A$ contains the edge $(z, c)$, or both the edges $(\alpha^\ast, c)$ and $(\beta^\ast, c)$. Finally, suppose $w \in V_A \backslash \{z, \alpha^\ast, \beta^\ast\}$. Then, by the construction of $G_A$, vertex $w$ is affected. By Lemma 3.8, $w \in D(c)$, so Lemma 3.5 implies that there is a path in $G$.
from \( c \) to \( w \) that contains only vertices in \( D(c) \). Hence, by construction, \( G_A \) contains a path from \( c \) to \( w \) that avoids \( \alpha^* \) and \( \beta^* \), so \( \alpha^* \) and \( \beta^* \) do not dominate \( w \). It remains to show that \( w \) is not dominated in \( G_A \) by \( c \) or another affected vertex \( v \). Let \((x, y)\) be the inserted edge. Without loss of generality, assume that \( c \prec x \). Since \( w \) is affected, there is a path \( \pi \) in \( G \) from \( y \) to \( w \) that satisfies Lemma 3.1. Then \( \pi \) does not contain any vertex in \( D[c, d(w)] \). Also, by the construction of \( G_A \), \( \pi \) corresponds to a path \( \pi_A \) in \( G_A \) from \( \beta^* \) to \( y \) that avoids any vertex in \( A \cap D[c, d(w)] \). Hence, \( w \) is not dominated by any vertex in \( A \cap D[c, d(w)] \). It remains to show that \( w \) is not dominated by any affected vertex \( v \) in \( A \setminus D[c, d(w)] \). Since both \( v \) and \( w \) are in \( D(c) \) and \( v \) is not an ancestor of \( w \) in \( D \), there is a path \( \pi' \) in \( G \) from \( c \) to \( w \) that avoids \( v \). By Lemma 3.5, \( \pi' \) contains only vertices in \( D(c) \). Then, by the construction of \( G_A \), \( \pi' \) corresponds to a path \( \pi_A' \) in \( G_A \) from \( c \) to \( w \) that avoids \( v \). Thus, \( v \) does not dominate \( w \) in \( G_A \).  

\[ \text{Lemma 3.13. Let } \nu \text{ and } \mu \text{, respectively, be the number of scanned vertices and their adjacent edges. Then, the derived affected flow graph } G_A \text{ has } \nu + 4 \text{ vertices, at most } \mu + 5 \text{ edges, and can be constructed in } O(\nu + \mu) \text{ time.} \]

\[ \text{Proof.} \text{ The bound on the number of vertices and edges in } G_A \text{ follows from the definition of the derived affected flow graph. Next, we consider the construction time of } G_A. \text{ Consider the edges entering the affected vertices. Let } w \text{ be an affected vertex, and let } (u, w) \neq (x, y) \text{ be an edge of } G'. \text{ Let } q \text{ be nearest ancestor } u \text{ in } C'(z). \text{ We distinguish two cases:} \]

- \( u \) is not scanned. In this case, we argue that \( q = c \). Indeed, it follows from the parent property of \( D \) and Lemma 3.8 that both \( u \) and \( w \) are descendants of \( c \) in \( D \). Since \( u \) is not scanned, no ancestor of \( u \) in \( D \) is affected, so \( u \) remains a descendant of \( c \) in \( D' \). Thus, \( q = c \).

- \( u \) is scanned. Then, by Lemma 3.2, \( q \) is the nearest affected ancestor of \( u \) in \( D \).

So we can construct the edges entering the affected vertices in \( G_A \) in two phases. In the first phase we traverse the descendants of each affected vertex \( q \) in \( D' \). At each descendant \( u \) of \( q \), we examine the edges leaving \( u \). When we find an edge \((u, w)\) with \( w \) affected, then we insert into \( G_A \) the edge \((q, w)\). In the second phase we examine the edges entering each affected vertex \( w \). When we find an edge \((u, w)\) with \( u \) not visited during the first phase (i.e., \( u \) was not scanned during the update of \( D \)), we insert into \( G_A \) the edge \((c, w)\). Note that during this construction we may insert the same edge multiple times, but this does not affect the correctness or running time of our overall algorithm. Since the descendants of an affected vertex are scanned, it follows that each phase runs in \( O(\nu + \mu) \) time.
Finally, we need to consider the inserted edge \((x, y)\). Let \(f\) be the nearest ancestor of \(x\) that is in \(C(z)\). Since \(y\) is affected, \(c \neq f\). Hence, we insert into \(G_A\) the edge \((\beta^*, y)\) if \(c <_\delta f\), and the edge \((\alpha^*, y)\) if \(f <_A c\). Note that \(f\) is found during the computation of \(z = \text{nca}(x, y)\), so this test takes constant time. \(\square\)

We use algorithm \texttt{DerivedLowHigh}, shown below, to order the vertices in \(C'(z)\) according to a low-high order of \(\zeta\) of \(G_A\). After computing \(G_A\), we construct two divergent spanning trees \(B_A\) and \(R_A\) of \(G_A\). For each vertex \(v \neq z\), if \((z, v)\) is an edge of \(G_A\), we replace the parent of \(v\) in \(B_A\) and \(R_A\), denoted by \(b_A(v)\) and \(r_A(v)\), respectively, by \(z\). Then we use algorithm \texttt{AuxiliaryLowHigh} to compute a low-high order \(\zeta\) of \(G_A\). Algorithm \texttt{AuxiliaryLowHigh} is a slightly modified version of a linear-time algorithm of [25, Section 6.1] to compute a low-high order. Our modified version computes a low-high order \(\zeta\) of \(G_A\) that is an augmentation of \(\delta_z\). To obtain such a low-high order, we need to assign to \(\alpha^*\) the lowest number in \(\zeta\) and to \(\beta^*\) the highest number in \(\zeta\). The algorithm works as follows. While \(G_A\) contains at least four vertices, we choose a vertex \(v \notin \{\alpha^*, \beta^*\}\) whose in-degree in \(G_A\) exceeds its number of children in \(B_A\) plus its number of children in \(R_A\) and remove it from \(G_A\). (From this choice of \(v\) we also have that \(v \neq z\).) Then we compute recursively a low-high order for the resulting flow graph, and insert \(v\) in an appropriate location, defined by \(b_A(v)\) and \(r_A(v)\).

\begin{algorithm}
\textbf{Algorithm 4: DerivedLowHigh}(\(z, A, \text{mark}\))
\begin{enumerate}
\item Compute the derived affected flow graph \(G_A = (V_A, E_A, z)\).
\item Compute two divergent spanning trees \(B_A\) and \(R_A\) of \(G_A\).
\item \textbf{foreach} vertex \(v \in V_A \setminus \{z, \alpha^*, \beta^*\}\) \textbf{do}
  \begin{enumerate}
  \item if \(\text{mark}(v) = \text{true}\) then set \(b_A(v) \leftarrow z\) and \(r_A(v) \leftarrow z\)
  \end{enumerate}
\itemInitialize a list of vertices \(\Lambda \leftarrow \emptyset\).
\itemCompute \(\Lambda \leftarrow \text{AuxiliaryLowHigh}(G_A, B_A, R_A, \Lambda)\).
\itemOrder the set of children \(C'(z)\) of \(z\) in \(D'\) according to \(\Lambda\).
\end{enumerate}
\end{algorithm}

Lemma 3.14. Algorithm \texttt{AuxiliaryLowHigh} is correct, that is, it computes a low-high order \(\zeta\) of \(G_A\), such that for all \(v \in V_A \setminus \{z, \alpha^*, \beta^*\}\), \(\alpha^* <_\zeta v <_\zeta \beta^*\).

\textit{Proof.} We first show that algorithm \texttt{AuxiliaryLowHigh} runs to completion, i.e., it selects every vertex \(v \in V_A \setminus \{z, \alpha^*, \beta^*\}\) at some execution of line 5. The recursive call in line 13 invokes algorithm \texttt{AuxiliaryLowHigh} on a sequence of smaller flow graphs \(G_A\). We claim that the following invariants hold for each such flow graph \(G_A\):

(i) the dominator tree \(D_A\) of \(G_A\) is flat;

(ii) the subgraphs \(B_A\) and \(R_A\) corresponding to \(G_A\) are divergent spanning trees of \(G_A\) rooted at \(z\);

(iii) for every \(v \neq z\), either \(b_A(v) = r_A(v) = z\) or \(b_A(v), r_A(v)\), and \(z\) are all distinct.

For the initial graph \(G_A\) the invariants hold by construction. Assume that the invariants hold on entry to line 5. Suppose, now, that line 5 chooses a vertex \(v \notin \{\alpha^*, \beta^*\}\). Since \(v\) has in-degree at most 2 in \(G_A\), the choice of \(v\) implies that it has at most one outgoing edge. Hence \(v\) is a leaf in either \(B_A\) or \(R_A\). If it is a leaf in both, deleting \(v\) and its incoming edges preserves all the invariants. Suppose \(v\) is a leaf in \(R_A\) but not \(B_A\). Then \(v\) has in-degree 2 in \(G_A\); that is, \(b_A(v) \neq r_A(v)\), which
Algorithm 5: AuxiliaryLowHigh($G_A, B_A, R_A, \Lambda$)

1. if $G_A$ contains only three vertices then
   2. set $\Lambda \leftarrow (\alpha^*, \beta^*)$
   3. return $\Lambda$
4. end
5. Let $v \notin \{\alpha^*, \beta^*\}$ be a vertex whose in-degree in $G_A$ exceeds its number of children in $B_A$ plus its number of children in $R_A$.
6. Delete $v$ and its incoming edges from $G_A$, $B_A$, and $R_A$.
7. if $v$ was not a leaf in $B_A$ then
   8. let $w$ be the child of $v$ in $B_A$; replace $b_A(w)$ by $b_A(v)$
9. end
10. else if $v$ was not a leaf in $R_A$ then
11. let $w$ be the child of $v$ in $R_A$; and replace $r_A(w)$ by $r_A(v)$
12. end
13. Call AuxiliaryLowHigh($G_A, B_A, R_A, \Lambda$) recursively for the new graph $G_A$.
14. if $b_A(v) = z$ then
15. insert $v$ anywhere between $\alpha^*$ and $\beta^*$ in $\Lambda$
16. end
17. else
18. insert $v$ just before $b_A(v)$ in $\Lambda$ if $r_A(v)$ is before $b_A(v)$ in $\Lambda$, just after $b_A(v)$ otherwise
19. end
20. return $\Lambda$

implies by (iii) that $b_A(v)$, $r_A(v)$, and $v$ are distinct siblings in $D_A$. Let $w$ be the child of $v$ in $B_A$. Since $r_A(w) \neq v, v, r_A(w)$, and $z$ are distinct by (iii). Also $r_A(w) \neq b_A(v)$, since $r_A(w) = b_A(v)$ would imply that $r_A(w)$ dominates $w$ by (ii). Finally, $b_A(v) \neq z$, since $b_A(v)$ is a sibling of $v$ and hence of $w$ in $D_A$. We conclude that replacing $b_A(w)$ by $b_A(v)$ in line 8 preserves (iii). This replacement preserves (i) since $v$ does not dominate $w$, it preserves (ii) since it removes $v$ from the path in $B_A$ from $s$ to $w$. Replacing $b_A(w)$ makes $v$ a leaf in $B_A$, after which its deletion preserves (i)-(iii).

Now we show that the invariants imply that line 5 can always choose a vertex $v$. All vertices in $V_A \setminus z$ are leaves in $D_A$. Let $X$ be the subset of $V_A$ that consists of the vertices $x$ such that $b_A(x) \neq r_A(x)$. Each vertex in $X$ has in-degree 2 in $G_A$, so there are $2|X|$ edges that enter a vertex in $X$. By invariant (iii), each edge leaving a vertex in $X$ enters a vertex in $X$. Invariant (iii) also implies that at least two edges enter $X$ from $V_A \setminus X$. Hence, there are at most $2(|X| - 1)$ edges that leave a vertex in $X$, so there must be a vertex $v$ in $X$ with out-degree at most 1. We claim that $v$ can be selected in line 5. First note that the in-degree of $v$ in $G_A$ exceeds its out-degree in $G_A$. If $v$ is a leaf in both $B_A$ and $R_A$ then it can be selected. If not, then $v$ must be a leaf in either $B_A$ or $R_A$, since otherwise its common child $w$ in $B_A$ and $R_A$ would violate (ii). Hence $v$ can be selected in this case also.

Finally, we claim that the computed order is low-high for $G_A$, such that $\alpha^*$ is first and $\beta^*$ is last in this order. The latter follows by the assignment in line 2. So the claim is immediate if $G_A$ has three vertices. Suppose, by induction, that this is true if $G_A$ has $k \geq 3$ vertices. Let $G_A$ have $k + 1$ vertices and let $v$ be the vertex chosen for deletion. The insertion position of $v$ guarantees that $v$ has the low-high property. All vertices in $G_A$ after the deletion of $v$ have the low-high property in the new $G_A \setminus z$ by the induction hypothesis, so they have the low-high property in the old $G_A$. 

14
with the possible exception of \( w \), one of whose incoming edges differs in the old and the new \( G_A \). Suppose \( b_A(w) \) differs; the argument is symmetric if \( r_A(w) \) differs. Now we have that \( v, w, b_A(v), \) and \( r_A(w) \) are distinct children of \( z \) in \( D_A \). Since \( w \) has the low-high property in the new \( G_A \), it occurs in \( \Lambda \) between \( r_A(w) \) and \( b_A(v) \). Insertion of \( v \) next to \( b_A(v) \) leaves \( w \) between \( r_A(w) \) and \( v \), so it has the low-high property in the old \( G_A \) as well.

The correctness of algorithm \textit{InsertEdge} follows from Lemmata 3.10, 3.11 and 3.14.

**Lemma 3.15.** Algorithm \textit{InsertEdge} is correct.

**Proof.** Let \( (G', D', \delta', \text{mark}', \text{low}', \text{high}') \) be the output of \textit{InsertEdge}\((G, D, \delta, \text{mark}, \text{low}, \text{high}, e)\). We only need to consider the case where both endpoints of the inserted edge \( e = (x, y) \) are reachable in \( G \). Let \( A \) be the set of affected vertices, and let \( z = nca(x, y) \). Also, let \( c \) be the child of \( z \) in \( D \) that is a common ancestor of all vertices in \( A \). We will show that the computed order \( \delta' \) is a low-high order of \( G' \) that agrees with \( \delta \). This fact implies that the arrays \( \text{mark}', \text{low}', \text{high}'\) were updated correctly, since their entries did not change for the vertices in \( V \setminus (A \cup \{c\}) \).

By construction, \( \delta' \) agrees with \( \delta \). Let \( \delta_z \) (resp., \( \delta'_z \)) be the restriction of \( \delta \) (resp., \( \delta' \)) to \( C(z) \) (resp., \( C'(z) \)). Then, by Lemma 3.14, \( \delta'_z \) is an augmentation of \( \delta_z \). So, by Lemmata 3.10 and 3.11, \( \delta' \) is a low-high order in \( G' \) for any vertex \( v \notin A \cup \{c\} \). Finally, Lemma 3.14 implies that \( \delta' \) is also a low-high order in \( G' \) for the vertices in \( A \cup \{c\} \).

**Theorem 3.16.** Algorithm \textit{InsertEdge} maintains a low-high order of a flow graph \( G \) with \( n \) vertices through a sequence of edge insertions in \( O(mn) \) total time, where \( m \) is the total number of edges in \( G \) after all insertions.

**Proof.** Consider the insertion of an edge \((x, y)\). If \( y \) was unreachable in \( G \) then we compute \( D \) and a low-high order in \( O(m) \) time. Throughout the whole sequence of \( m \) insertions, such an event can happen \( O(n) \) times, so all insertions to unreachable vertices are handled in \( O(mn) \) total time.

Now we consider the cost of executing \textit{InsertEdge} when both \( x \) and \( y \) are reachable in \( G \). Let \( \nu \) be the number of scanned vertices, and let \( \mu \) be the number of their adjacent edges. We can update the dominator tree and locate the affected vertices (line 8) in \( O(\nu + \mu + n) \) time \cite{23}. Computing \( z = nca(x, y) \) in line 7 takes \( O(n) \) time just by using the parent function \( d \) of \( D \). Lines 9–10 and 12 are also executed in \( O(n) \) time. The for loop in lines 13–16 takes \( O(\nu + \mu) \) since we only need to examine the scanned edges. (Variables \( \text{low}(c) \) and \( \text{high}(c) \) need to be updated only if there is a scanned edge entering \( c \).) It remains to account for time to compute \( G_A \) and a low-high order of it. From Lemma 3.13, the derived affected flow graph can be constructed in \( O(\nu + \mu) \) time. In algorithm \textit{AuxiliaryLowHigh}, we represent the list \( \Lambda \) with the off-line dynamic list maintenance data structure of \cite{25}, which supports insertions (in a given location) and order queries in constant time.

With this implementation, \textit{AuxiliaryLowHigh} runs in linear-time, that is \( O(\nu + \mu) \). So \textit{InsertEdge} runs in \( O(\nu + \mu + n) \) time. The \( O(n) \) term gives a total cost of \( O(mn) \) for the whole sequence of \( m \) insertions. We distribute the remaining \( O(\nu + \mu) \) cost to the scanned vertices and edges, that is \( O(1) \) per scanned vertex or edge. Since the depth in \( D \) of every scanned vertex decreases by at least one, a vertex and an edge can be scanned at most \( O(n) \) times. Hence, each vertex and edge can contribute at most \( O(n) \) total cost through the whole sequence of \( m \) insertions. The \( O(mn) \) bound follows.
4 Applications of incremental low-high orders

4.1 Strongly divergent spanning trees and path queries

We can use the arrays mark, low, and high to maintain a pair of strongly divergent spanning trees, $B$ and $R$, of $G$ after each update. Recall that $B$ and $R$ are strongly divergent if for every pair of vertices $v$ and $w$, we have $B[s, v] \cap R[s, w] = D[s, v] \cap D[s, w]$ or $R[s, v] \cap B[s, w] = D[s, v] \cap D[s, w]$. Moreover, we can construct $B$ and $R$ so that they are also edge-disjoint except for the bridges of $G$. A bridge of $G$ is an edge $(u, v)$ that is contained in every path from $s$ to $v$. Let $b(v)$ (resp., $r(v)$) denote the parent of a vertex $v$ in $B$ (resp., $R$). To update $B$ and $R$ after the insertion of an edge $(x, y)$, we only need to update $b(v)$ and $r(v)$ for the affected vertices $v$, and possibly for their common ancestor $c$ that is a child of $z = nca(x, y)$ from Lemma 3.8. We can update $b(v)$ and $r(v)$ of each vertex $v \in A \cup \{c\}$ as follows: set $b(v) \leftarrow d(v)$ if $low(v) = null$, $b(v) \leftarrow low(v)$ otherwise; set $r(v) \leftarrow d(v)$ if $high(v) = null$, $r(v) \leftarrow high(v)$ otherwise. If the insertion of $(x, y)$ does not affect $y$, then $A = \emptyset$ but we may still need to update $b(y)$ and $r(y)$ if $x \notin D(y)$ in order to make $B$ and $R$ maximally edge-disjoint. Note that in this case $z = d(y)$, so we only need to check if both $low(y)$ and $high(y)$ are null. If they are, then we set $low(y) \leftarrow x$ if $x <_{\delta} y$, and set $high(y) \leftarrow x$ otherwise. Then, we can update $b(y)$ and $r(y)$ as above.

Now consider a query that, given two vertices $v$ and $w$, asks for two maximally vertex-disjoint paths, $\pi_{sv}$ and $\pi_{sw}$, from $s$ to $v$ and from $s$ to $w$, respectively. Such queries were used in [47] to give a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph. If $v <_{\delta} w$, then we select $\pi_{sv} \leftarrow B[s, v]$ and $\pi_{sw} \leftarrow R[s, w]$; otherwise, we select $\pi_{sv} \leftarrow R[s, v]$ and $\pi_{sw} \leftarrow B[s, w]$. Therefore, we can find such paths in constant time, and output them in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. Similarly, for any two query vertices $v$ and $w$, we can report a path $\pi_{sv}$ from $s$ to $v$ that avoids $w$. Such a path exists if and only if $w$ does not dominate $v$, which we can test in constant time using the ancestor-descendant relation in $D$ [44]. If $w$ does not dominate $v$, then we select $\pi_{sv} \leftarrow B[s, v]$ if $v <_{\delta} w$, and select $\pi_{sv} \leftarrow R[s, v]$ if $w <_{\delta} v$.

4.2 Fault tolerant reachability

Baswana et al. [6] study the following reachability problem. We are given a flow graph $G = (V, E, s)$ and a spanning tree $T = (V, E_T)$ rooted at $s$. We call a set of edges $E'$ valid if the subgraph $G' = (V, E_T \cup E', s)$ of $G$ has the same dominators as $G$. The goal is to find a valid set of minimum cardinality. As shown in [26], we can compute a minimum-size valid set in $O(m)$ time, given the dominator tree $D$ and a low-high order of $\delta$ of it. We can combine the above construction with our incremental low-high algorithm to solve the incremental version of the fault tolerant reachability problem, where $G$ is modified by edge insertions and we wish to compute efficiently a valid set for any query spanning tree $T$. Let $t(v)$ be the parent of $v$ in $T$. Our algorithm maintains, after each edge insertion, a low-high order $\delta$ of $G$, together with the mark, low, and high arrays. Given a query spanning tree $T = (V, E_T)$, we can compute a valid set of minimum cardinality $E'$ as follows. For each vertex $v \neq s$, we apply the appropriate one of the following cases: (a) If $t(v) = d(v)$ then we do not insert into $E'$ any edge entering $v$. (b) If $t(v) \neq d(v)$ and $v$ is marked then we insert $(d(v), v)$ into $E'$. (c) If $v$ is not marked then we consider the following subcases: If $t(v) >_{\delta} v$, then we insert into $E'$ the edge $(x, v)$ with $x = low(v)$. Otherwise, if $t(v) <_{\delta} v$, then we insert into $E'$ the edge $(x, v)$ with $x = high(v)$. Hence, can update the minimum valid set in $O(mn)$ total time.

We note that the above construction can be easily generalized for the case where $T$ is forest, i.e., when $E_T$ is a subset of the edges of some spanning tree of $G$. In this case, $t(v)$ can be null for some vertices $v \neq s$. To answer a query for such a $T$, we apply the previous construction with the following modification when $t(v)$ is null. If $v$ is marked then we insert $(d(v), v)$ into $E'$, as in
case (b). Otherwise, we insert both edges entering \( v \) from \( \text{low}(v) \) and \( \text{high}(v) \). In particular, when \( E_T = \emptyset \), we compute a subgraph \( G' = (V,E',s) \) of \( G \) with minimum number of edges that has the same dominators as \( G \). This corresponds to the case \( k = 1 \) in [7].

4.3 Sparse certificate for 2-edge-connectivity

Let \( G = (V,E) \) be a strongly connected digraph. We say that vertices \( u,v \in V \) are 2-edge-connected if there are two edge-disjoint directed paths from \( u \) to \( v \) and two edge-disjoint directed paths from \( v \) to \( u \). (A path from \( u \) to \( v \) and a path from \( v \) to \( u \) need not be edge-disjoint.) A 2-edge-connected block of a digraph \( G = (V,E) \) is defined as a maximal subset \( B \subseteq V \) such that every two vertices in \( B \) are 2-edge-connected. If \( G \) is not strongly connected, then its 2-edge-connected blocks are the 2-edge-connected blocks of each strongly connected component of \( G \). A sparse certificate for the 2-edge-connected blocks of \( G \) is a spanning subgraph \( C(G) \) of \( G \) that has \( O(n) \) edges and maintains the same 2-edge-connected blocks as \( G \). Sparse certificates of this kind allow us to speed up computations, such as finding the actual edge-disjoint paths that connect a pair of vertices (see, e.g., [39]). The 2-edge-connected blocks and a corresponding sparse certificate can be computed in \( O(mn) \) time [21]. An incremental algorithm for maintaining the 2-edge-connected blocks is presented in [24]. This algorithm maintains the dominator tree of \( G \), with respect to an arbitrary start vertex \( s \), and of its reversal \( G^R \), together with the auxiliary components of \( G \) and \( G^R \), defined next.

Recall that an edge \( (u,v) \) is a bridge of a flow graph \( G \) with start vertex \( s \) if all paths from \( s \) to \( v \) include \( (u,v) \). After deleting from the dominator tree \( D \) the bridges of \( G \), we obtain the bridge decomposition of \( D \) into a forest \( D \). For each root \( r \) of a tree in the bridge decomposition \( D \) 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 \( D_r \). The edge set \( E_r \) contains all the edges of \( G \) among the vertices of \( V_r \), referred to as ordinary edges, and a set of auxiliary edges, which are obtained by contracting vertices in \( V \setminus V_r \), as follows. Let \( v \) be a vertex in \( V_r \) that has a child \( w \) in \( V \setminus V_r \). Note that \( w \) is a root in the bridge decomposition \( D \) of \( D \). For each such child \( w \) of \( v \), we contract \( w \) and all its descendants in \( D \) into \( v \). The auxiliary components of \( G \) are the strongly connected components of each auxiliary graph \( G_r \).

We sketch how to extend the incremental algorithm of [24] so that it also maintains a sparse certificate \( C(G) \) for the 2-edge-connected components of \( G \), in \( O(mn) \) total time. It suffices to maintain the auxiliary components in \( G \) and \( G^R \), and two maximally edge-disjoint divergent spanning trees for each of \( G \) and \( G^R \). We can maintain these divergent spanning trees as described in Section 4.1. To identify the auxiliary components, the algorithm of [24] uses, for each auxiliary graph, an incremental algorithm for maintaining strongly connected components [8]. It is easy to extend this algorithm so that it also computes \( O(n) \) edges that define these strongly connected components. The union of these edges and of the edges in the divergent spanning trees are the edges of \( C(G) \).

5 2-vertex-connected spanning subgraph

Let \( G = (V,E) \) be a strongly connected digraph. A vertex \( x \) of \( G \) is a strong articulation point if \( G \setminus x \) is not strongly connected. A strongly connected digraph \( G \) is 2-vertex-connected if it has at least three vertices and no strong articulation points [19, 31]. Here we consider the problem of approximating a smallest 2-vertex-connected spanning subgraph (2VCSS) of \( G \). This problem is NP-hard [17]. We show that algorithm LH-Z (given below), which uses low-high orders, achieves a linear-time 2-approximation for this problem. The best previous approximation ratio achievable in
Algorithm 6: LH-Z(G)

Input: 2-vertex-connected digraph $G = (V, E)$

Output: 2-approximation of a smallest 2-vertex-connected spanning subgraph $H = (V, E_H)$ of $G$

1. Choose an arbitrary vertex $s$ of $G$ as start vertex.
2. Compute a strongly connected spanning subgraph $H = (V \setminus s, E_H)$ of $G \setminus s$.
3. Set $H ← (V, E_H)$.
4. Compute a low-high order $δ$ of flow graph $G$ with start vertex $s$.
5. foreach vertex $v ≠ s$ do
   6. if there are two edges $(u, v)$ and $(w, v)$ in $E_H$ such that $u < δ v$ and $v < δ w$ then
      7. do nothing
   else if there is no edge $(u, v) ∈ E_H$ such that $u < δ v$ then
      8. find an edge $e = (u, v) ∈ E$ with $u < δ v$
      9. set $E_H ← E_H ∪ \{e\}$
   end
   10. else if there is no edge $(w, v) ∈ E_H$ such that $v < δ w$ then
      11. find an edge $e = (w, v) ∈ E$ with $v < δ w$ or $w = s$
      12. set $E_H ← E_H ∪ \{e\}$
   end
13. end
14. Execute the analogous steps of lines 4–17 for the reverse flow graph $G^R$ with start vertex $s$.
15. return $H = (V, E_H)$

linear-time was 3 [20], so we obtain a substantial improvement. The best approximation ratio for 2VCSS is $3/2$, and is achieved by the algorithm of Cheriyan and Thurimella [10] in $O(m^2)$ time, or in $O(m√n + n^2)$ by a combination of [10] and [20]. Computing small spanning subgraphs is of particular importance when dealing with large-scale graphs, e.g., with hundreds of million to billion edges. In this framework, one big challenge is to design linear-time algorithms, since algorithms with higher running times might be practically infeasible on today’s architectures. Let $G = (V, E)$ be a strongly connected digraph. In the following, we denote by $G^R = (V, E^R)$ the reverse digraph of $G$ that results from $G$ after reversing all edge directions.

Lemma 5.1. Algorithm LH-Z computes a 2-vertex-connected spanning subgraph of $G$.

Proof. We need to show that the computed subgraph $H$ is 2-vertex-connected. From [31], we have that a digraph $H$ is 2-vertex connected if and only if it satisfies the following property: For an arbitrary start vertex $s ∈ V$, flow graphs $H = (V, E, s)$ and $H^R = (V, E^R, s)$ have flat dominator trees, and $H \setminus s$ is strongly connected. The digraph $H$ computed by algorithm LH-Z satisfies the latter condition because of line 2. It remains to show that $H$ has flat dominator tree. The same argument applies for $H^R$, thus completing the proof. Let $δ$ be the low-high order $δ$ of $G$, computed in line 3. We argue that after the execution of the for loop in lines 5–17, $δ$ is also a low-high order for all vertices in $H$. Consider an arbitrary vertex $v ≠ s$. Let $(x, v)$ be an edge entering $v$ in the strongly connected spanning subgraph of $G$ computed in line 2. If $x > δ v$, then, by the definition of $δ$, there is at least one edge $(y, v) ∈ E$ such that $y < δ v$. Hence, after the execution of the for loop for $v$, the edge set $E_H$ will contain at least two edges $(u, v)$ and $(w, v)$ such that $u < δ v < δ w$. On the other hand, if $x < δ v$, then the definition of $δ$ implies that there an edge $(y, v) ∈ E$ such that
Table 1: Real-world graphs used in the experiments, sorted by the file size of their largest SCC. We used both the largest SCC and the some of the 2VCCs (inside the largest SCC) in our experiments.

| Graph                | Largest SCC | 2VCCs | Type            |
|----------------------|-------------|-------|-----------------|
| rome99               | 3352/8855   | 2249/6467 | road network    |
| twitter-higgs-retweet| 13086/63537| 1099/9290 | twitter         |
| enron                | 8271/147353 | 4441/123527 | enron mails     |
| web-NotreDame        | 48715/267647 | 1409/7279 | web             |
| soc-Epinions1        | 32220/442768 | 1416/13226 | web             |
| amazon-302           | 241761/1131217 | 17117/395183 | trust network   |
| wikiTalk             | 111878/1477665 | 5179/12897 | wiki communications |
| web-Stanford         | 150475/1576157 | 10893/162295 | web             |
| web-Amazon-302       | 95320/1340351 | 276049/241663 | co-purchase     |
| web-BerkStan         | 334857/4523232 | 1106/8206 | web             |
| web-Enron           | 8271/147353 | 4441/123527 | enron mails     |
| web-NotreDame        | 48715/267647 | 1409/7279 | web             |
| web-Amazon-601       | 395230/3301051 | 276049/241663 | co-purchase     |
| web-BerkStan         | 334857/4523232 | 1106/8206 | web             |

6 Empirical Analysis

For the experimental evaluation we use the graph datasets shown in Table 1. We wrote our implementations in C++, using g++ v.4.6.4 with full optimization (flag -O3) to compile the code. We report the running times on a GNU/Linux machine, with Ubuntu (12.04LTS): a Dell PowerEdge
R715 server 64-bit NUMA machine with four AMD Opteron 6376 processors and 128GB of RAM memory. Each processor has 8 cores sharing a 16MB L3 cache, and each core has a 2MB private L2 cache and 2300MHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the \texttt{getrusage} function, averaged over ten different runs. In Table \ref{table:stats} we can see some statistics about the real-world graphs we used in our experimental evaluation.

### 6.1 Incremental low-high order.

We compare the performance of four algorithms. As a baseline, we use a static low-high order algorithm from \cite{25} based on an efficient implementation of the Lengauer-Tarjan algorithm for computing dominators \cite{35} from \cite{27}. Our baseline algorithm, SLT, constructs, as intermediary, two divergent spanning trees. After each insertion of an edge \((x,y)\), SLT recomputes a low-high order if \(x\) is reachable. An improved version of this algorithm, that we refer to as SLT-NCA, tests if the insertion of \((x,y)\) affects the dominator tree by computing the nearest common ancestor of \(x\) and \(y\). If this is the case, then SLT-NCA recomputes a low-high order as SLT. The other two algorithms are the ones we presented in Section 3. For our simple algorithm, DBS-DST, we extend the incremental dominators algorithm DBS of \cite{23} with the computation of two divergent spanning trees and a low-high order, as in SLT. Algorithm DBS-DST applies these computations on a sparse subgraph of the input digraph that maintains the same dominators. Finally, we tested an implementation of our more efficient algorithm, DBS-AUX, that updates the low-high order by computing a local low-high order of an auxiliary graph.

We compared the above incremental low-high order algorithms in two different field tests. In the first one, we considered 2-vertex connected graphs, and we dynamized them in the following manner: we removed a percentage of edges (i.e., 5\%, 10\%, and 20\% respectively), selected uniformly at random, that were incrementally added to the graph. Note that during the execution of the algorithms some vertices may be unreachable at first. Also, at the end of all insertions, the final graph has flat dominator tree. In Figure 4 (top) we can see that the algorithms are well
in Figure 4 (bottom), but the difference is bigger: we note a bigger gap of more than two orders of magnitude, in particular, between DBS-AUX and the couple SLT-NCA and DBS-DST. This is expected because, unlike the first test, here all edges connect already reachable vertices. This means that DBS-DST and DBS-AUX do not execute a full restart for any of these insertions. The total running times are given in Table 3.

6.2 2-vertex-connected spanning subgraph.

In this experimental evaluation we compared four algorithms for computing the (approximated) smallest 2-vertex-connected spanning subgraph. Specifically, we tested two algorithms from [20], FAST which computes a 3-approximation in linear-time by using divergent spanning trees, and FAST-CT which combines FAST with the 3/2-approximation algorithm of Cheriyan and Thurimella [10]. In the experiments reported in [20], the former algorithm achieved the fastest running times, while the latter the best solution quality. We compare these algorithms against our new algorithm LH-Z of Section 5, and a new hybrid algorithm LH-Z-CT, that combines LH-Z with the algorithm of Cheriyan and Thurimella [10].

Algorithm LH-Z-CT works as follows. First, it computes a 1-matching $M$ in the input graph $G$ [16], using bipartite matching as in [10]. Let $H$ be the subgraph of $G \setminus s$, for arbitrary start vertex $s$, that contains only the edges in $M$. We compute the strongly connected components $C_1, \ldots, C_k$
### Table 2: Running times of the plot shown in Figure 4 (top).

| Graph     | nodes   | starting edges | final edges | SLT    | SLT-NCA | DBS-AUX | DBS-DST |
|-----------|---------|----------------|-------------|--------|---------|---------|---------|
| rome05    | 2249    | 6144           | 6467        | 0.2109 | 0.1202  | 0.2426  | 0.3190  |
| rome10    | 2249    | 5820           | 6467        | 0.7346 | 0.2317  | 0.2426  | 0.3190  |
| rome20    | 2249    | 5174           | 6467        | 0.7279 | 0.6438  | 0.2317  | 0.3190  |
| twitter05 | 1099    | 8826           | 9290        | 0.3283 | 0.0511  | 0.2426  | 0.3190  |
| twitter10 | 1099    | 8361           | 9290        | 0.9968 | 0.0859  | 0.0649  | 0.2794  |
| twitter20 | 1099    | 7432           | 9290        | 2.0674 | 0.1982  | 0.0180  | 0.0709  |
| NotreDame05 | 1416   | 12565          | 13226       | 0.2129 | 0.0027  | 0.0034  | 0.0033  |
| NotreDame10 | 1416   | 11903          | 13226       | 0.2129 | 0.0027  | 0.0034  | 0.0033  |
| NotreDame20 | 1416   | 10581          | 13226       | 0.1973 | 0.0144  | 0.0045  | 0.0033  |
| enron05   | 4441    | 117351         | 123527      | 51.54  | 37.42   | 7.92    | 26.52   |
| enron10   | 4441    | 111174         | 123527      | 214.21 | 180.22  | 25.09   | 45.18   |
| enron20   | 4441    | 98822          | 123527      | 353.17 | 297.78  | 25.09   | 45.18   |
| webStanford05 | 5179  | 123402         | 129897      | 42.17  | 37.28   | 21.10   | 93.12   |
| webStanford10 | 5179 | 116907         | 129897      | 51.28  | 41.46   | 21.10   | 93.12   |
| webStanford20 | 5179 | 103918         | 129897      | 51.61  | 44.87   | 21.10   | 93.12   |
| Amazon05  | 55414   | 229580         | 241663      | 185.86 | 121.26  | 26.54   | 251.87  |
| Amazon10  | 55414   | 217497         | 241663      | 51.28  | 41.46   | 21.10   | 93.12   |
| Amazon20  | 55414   | 193343         | 241663      | 51.61  | 44.87   | 21.10   | 93.12   |
| WikiTalk05 | 49430  | 1192153        | 1254898     | 150.26 | 113.94  | 20.24   | 90.41   |
| WikiTalk10 | 49430  | 1129408        | 1254898     | 248.46 | 247.60  | 17.59   | 45.71   |
| WikiTalk20 | 49430  | 1003918        | 1254898     | 516.82 | 500.58  | 45.91   | 99.60   |

### Table 3: Running times of the plot shown in Figure 4 (bottom).

| Graph     | nodes   | starting edges | final edges | SLT    | SLT-NCA | DBS-AUX | DBS-DST |
|-----------|---------|----------------|-------------|--------|---------|---------|---------|
| rome05    | 3352    | 8855           | 9298        | 0.662  | 0.185   | 0.002   | 0.079   |
| rome10    | 3352    | 8855           | 9741        | 1.065  | 0.366   | 0.005   | 0.147   |
| rome20    | 3352    | 8855           | 10626       | 2.742  | 0.410   | 0.008   | 0.259   |
| twitter05 | 13086   | 63537          | 66714       | 26.59  | 9.94    | 0.073   | 4.29    |
| twitter10 | 13086   | 63537          | 69891       | 55.49  | 41.46   | 0.121   | 8.73    |
| twitter20 | 13086   | 63537          | 76244       | 96.32  | 44.87   | 0.186   | 11.14   |
| enron05   | 8271    | 147353         | 154721      | 82.60  | 11.99   | 0.069   | 1.72    |
| enron10   | 8271    | 147353         | 162088      | 180.22 | 25.09   | 0.095   | 3.97    |
| enron20   | 8271    | 147353         | 176824      | 353.17 | 20.19   | 0.106   | 4.95    |
| NotreDame05 | 48715  | 267647         | 281092      | 785.02 | 375.36  | 0.706   | 234.76  |
| NotreDame10 | 48715 | 267647         | 294142      | 1691.05| 610.29  | 0.791   | 359.02  |
| NotreDame20 | 48715 | 267647         | 321176      | 3593.09| 1168.5  | 1.319   | 585.80  |
| Amazon05  | 241761  | 1131217        | 118778      | >24h   | 6386.97 | 26.54   | 3694.69 |
| Amazon10  | 241761  | 1131217        | 1244339     | >24h   | 11905.7 | 45.18   | 5628.51 |
| Amazon20  | 241761  | 1131217        | 1357460     | >24h   | 15871   | 60.19   | 915.7   |
| WikiTalk05 | 111878 | 1477665        | 1551548     | >24h   | 341.28  | 10.36   | 1157.84 |
| WikiTalk10 | 111878 | 1477665        | 1629432     | >24h   | 530.51  | 14.51   | 1666.28 |
| WikiTalk20 | 111878 | 1477665        | 1773198     | >24h   | 729.67  | 19.58   | 2124.6  |
| webStanford05 | 150475 | 1576157        | 1654065     | >24h   | 840.03  | 7.02    | 2285.55 |
| webStanford10 | 150475 | 1576157        | 1734773     | >24h   | 1150.4  | 13.73   | 3749.12 |
| webStanford20 | 150475 | 1576157        | 1891388     | >24h   | 15792.1 | 12.09   | 5381.12 |
in $H$, and form a contracted version $G'$ of $G$ as follows. For each strongly connected component $C_i$ of $H$, we contract all vertices in $C_i$ into a representative vertex $u_i \in C_i$. (Contractions are performed by union-find [45] and merging lists of out-edges of $G$.) Then, we execute the linear-time algorithm of Zhao et al. [49] to compute a strongly connected spanning subgraph of $G'$, and store the original edges of $G$ that correspond to the selected edges by the Zhao et al. algorithm. Let $Z$ be this set of edges. We compute a low-high order of $G$ with root $s$, and use it in order to compute a 2-vertex-connected spanning subgraph $W$ of $G$ using as many edges from $Z$ and $M$ as possible, as in LH-Z. Then, we run the filtering phase of Cheriyan and Thurimella. For each edge $(x,y)$ of $W$ that is not in $M$, we test if $x$ has two vertex-disjoint paths to $y$ in $W \setminus (x,y)$. If it does, then we set $W \leftarrow W \setminus (x,y)$. We remark that, similarly to FAST-CT, LH-Z preserves the 3/2 approximation guarantee of the Cheriyan-Thurimella algorithm for $k = 2$ and improves its running time from $O(m^2)$ to $O(m\sqrt{n} + n^2)$, for a digraph with $n$ vertices and $m$ arcs. In our implementation, the bipartite matching is computed via max-flow, using an implementation of the Goldberg-Tarjan push-relabel algorithm [28] from [13], which is very fast in practice. (This implementation was provided by the authors of [13].)

In Figure 5 (top) we can see the running times of the four algorithms. (See also Table 4.) It is easy to observe that the algorithms belong to two distinct classes, with FAST and LH-Z being faster than the other two by approximately five orders of magnitude. In the bottom part of Figure 5 we can see the relative size of the smallest spanning subgraph computed by the four algorithms. In all of our experiments, the smallest subgraph was the one computed by our new hybrid algorithm LH-Z-CT. One the other hand, on average LH-Z is only twice as slow as FAST but improves the solution quality by more than 13%. Summing up, if one wants a fast and good solution LH-Z is the right choice.

**Acknowledgement.** We would like to thank Bob Tarjan for valuable comments and suggestions.

**References**

[1] A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014.

| Graph            | Size  | FAST | LH-Z | LH-Z-CT | FAST-CT |
|------------------|-------|------|------|---------|---------|
|                  | nodes | edges| time | edges   | time    | edges   |
| rome99           | 2249  | 6467 | 0.003581 | 5691 | 0.004513 | 5370 | 2.099891 | 4837 | 5.042213 | 5057 |
| web-NotreDame1   | 1409  | 6856 | 0.002609 | 3796 | 0.001211 | 3356 | 0.927683 | 3029 | 2.119423 | 3153 |
| web-NotreDame2   | 1462  | 7279 | 0.002784 | 3949 | 0.005436 | 3545 | 0.999031 | 3189 | 2.262008 | 3300 |
| web-BerkStan1    | 1106  | 8206 | 0.001753 | 3423 | 0.003930 | 2795 | 0.525704 | 2433 | 0.949573 | 2440 |
| twitter-higgs-retweet | 1099 | 9290 | 0.003182 | 4687 | 0.005515 | 3990 | 0.751014 | 3560 | 2.143551 | 3768 |
| web-BerkStan2    | 4927  | 7279 | 0.002784 | 3949 | 0.005436 | 3545 | 0.999031 | 3189 | 2.262008 | 3300 |
| web-Stanford     | 5179  | 12987| 0.022056 | 3423 | 0.003930 | 2795 | 0.525704 | 2433 | 0.949573 | 2440 |
| Amazon-302       | 55414 | 241663| 0.118040 | 164979 | 0.164081 | 141467 | 2986.022813 | 123095 | 3935.135495 | 132847 |
| web-Google       | 77480 | 840829| 0.182113 | 256055 | 0.401247 | 221327 | 7668.066338 | 191616 | 20207.08225 | 211529 |
| WikiTalk         | 49430 | 1254898| 0.338172 | 176081 | 0.548573 | 161128 | 5883.002974 | 138030 | 4770.705853 | 143958 |
| Amazon-601       | 276049| 2461072| 0.977274 | 932989 | 1.607812 | 744345 | 140894.0119 | 612760 | 298775.2092 | 688159 |

Table 4: Running times and number of edges in the resulting 2-vertex-connected spanning subgraph; plots shown in Figure 5.
[2] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology, 230*(3):351–358, 2004.

[3] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing, 28*(6):2117–32, 1999.

[4] S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.

[5] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.

[6] S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015.

[7] S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. 48th ACM Symp. on Theory of Computing*, pages 509–518, 2016. To appear.

[8] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms, 12*(2):14:1–14:22, December 2015.

[9] 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.

[10] J. Cheriyan and R. Thurimella. Approximating minimum-size $k$-connected spanning subgraphs via matching. *SIAM J. Comput.*, 30(2):528–560, 2000.

[11] S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theor. Comput. Sci.*, 203:69–90, August 1998.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[13] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS’11)*, 2011.

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

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

[16] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM, 38*:815–853, 1991.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
[18] K. Gargi. A sparse algorithm for predicated global value numbering. *SIGPLAN Not.*, 37(5):45–56, May 2002.

[19] L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

[20] L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

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

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

[23] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012. Full version: *CoRR*, abs/1604.02711.

[24] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *Proc. 43nd Int'l. Coll. on Automata, Languages, and Programming*, 2016. To appear.

[25] L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015.

[26] L. Georgiadis and R. E. Tarjan. Addendum to “Dominator tree certification and divergent spanning trees”. *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016.

[27] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

[28] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35:921–940, October 1988.

[29] M. Gomez-Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

[30] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015.

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

[32] R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015.

[33] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164 – 172, 2016.
[34] S. Khuller, B. Raghavachari, and N. E. Young. Approximating the minimum equivalent digraph. *SIAM J. Comput.*, 24(4):859–872, 1995. Announced at *SODA 1994*, 177-186.

[35] 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.

[36] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD ’10, pages 115–124, 2010.

[37] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[38] M. Mowbray and A. Lain. Dominator-tree analysis for distributed authorization. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS ’08, pages 101–112, New York, NY, USA, 2008. ACM.

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

[40] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

[41] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 287–296, 1994.

[42] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.

[43] V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997.

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

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

[46] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[47] T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012.

[48] J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012.

[49] L. Zhao, H. Nagamochi, and T. Ibaraki. A linear time $5/3$-approximation for the minimum strongly-connected spanning subgraph problem. *Information Processing Letters*, 86(2):63–70, 2003.